**PONTIFICIA‌ ‌UNIVERSIDAD‌ ‌  
CATÓLICA‌ ‌DEL‌ ‌PERÚ**

**Escuela de Posgrado**



‌

Proyecto de Tesis

Una máquina virtual para el Wolfram Language *pattern matcher*

Tesis para obtener el grado académico de Maestro en Ingeniería Informática con Mención en Ciencias de la Computación que presenta:

***Héctor Daniel Sánchez Domínguez***

‌ ‌

Asesor:

***Dr. Andrés Melgar Sasieta***

Lima, 2025

Índice

Table of Contents

[Problemática 4](#_Toc196170674)

[1.1 Introducción 4](#_Toc196170675)

[1.2 Definición del Problema 4](#_Toc196170676)

[1.2.1 Descripción de la problemática 4](#_Toc196170677)

[1.2.2 Problema seleccionado 4](#_Toc196170678)

[1.3 Objetivos 5](#_Toc196170679)

[1.3.1 Objetivo general 5](#_Toc196170680)

[1.3.2 Objetivos específicos 5](#_Toc196170681)

[1.4 Resultados Esperados 5](#_Toc196170682)

[1.5 Métodos y Procedimientos 7](#_Toc196170683)

[1.6 Justificación 9](#_Toc196170684)

[1.7 Delimitación y Alcance 9](#_Toc196170685)

[2 Marco Conceptual 10](#_Toc196170686)

[2.1 Introducción 10](#_Toc196170687)

[2.2 *Wolfram Language* (WL) [2] 10](#_Toc196170688)

[2.3 *Pattern matching* [1] 10](#_Toc196170689)

[2.4 *(Process) Virtual machine* (VM) 10](#_Toc196170690)

[2.5 *Bytecode* 10](#_Toc196170691)

[2.6 *Instruction Set Architecture* (ISA) 10](#_Toc196170692)

[2.7 Compilación 10](#_Toc196170693)

[2.8 *Intermediate Representation* (IR) 10](#_Toc196170694)

[2.9 *Just-in-Time compilation* (JIT) 10](#_Toc196170695)

[*2.10* *Static Analysis* 10](#_Toc196170696)

[*2.11* *Tail-Code Optimization* (TCO) 10](#_Toc196170697)

[*2.12* *Memory Management* 10](#_Toc196170698)

[*2.13* *Memory Allocation* 10](#_Toc196170699)

[*2.14* *Memory Pooling* 10](#_Toc196170700)

[*2.15* *Debugging* 10](#_Toc196170701)

[*2.16* *Parser* 11](#_Toc196170702)

[3 Revisión del Estado del Arte 12](#_Toc196170703)

[3.1 Introducción 12](#_Toc196170704)

[3.2 Objetivos de revisión 12](#_Toc196170705)

[3.3 Preguntas de revisión 12](#_Toc196170706)

[3.4 Estrategia de Búsqueda 12](#_Toc196170707)

[3.4.1 Motores de búsqueda a usar 12](#_Toc196170708)

[3.4.2 Cadenas de búsqueda a usar 12](#_Toc196170709)

[3.4.3 Documentos encontrados 13](#_Toc196170710)

[3.5 Formulario de extracción de datos 14](#_Toc196170711)

[3.6 Resultados de la revisión 15](#_Toc196170712)

[3.7 Conclusiones 16](#_Toc196170713)

[4 Referencias 17](#_Toc196170714)

# Problemática

## Introducción

El *pattern matching* (coincidencia de patrones) [1] es un pilar fundamental del lenguaje de programación *Wolfram Language* (WL) [2], habilitando funciones esenciales como MatchQ, Replace, y programación basada en reglas. Sin embargo, su implementación actual, basada en recursión y gestión de memoria ad-hoc, presenta limitaciones críticas en **escalabilidad**, **rendimiento** y **extensibilidad**. Este proyecto propone el **diseño e implementación de una máquina virtual (VM)** [3] especializada para modernizar el *pattern matching*, combinando técnicas de compilación (bytecode, las instrucciones o la representación intermedia de los patrones), optimización de instrucciones, ejecución de instrucciones y manejo eficiente de memoria.

## Definición del Problema

### Descripción de la problemática

El sistema actual de *pattern matching* [1] en Wolfram Language enfrenta tres desafíos técnicos principales, respaldados por evidencia en la literatura y análisis del código fuente:

1. Recursión no optimizada:

La implementación actual utiliza recursión sin *Tail-Call Optimization* (TCO) [4], causando *stack overflows* en patrones profundos. Esto limita significativamente el manejo de estructuras complejas y genera bajo rendimiento de *matching* en patrones profundos y complejos. La situación ideal sería una implementación iterativa con gestión explícita del *stack* O(1) que evite estos límites.

1. Arquitectura monolítica:

Las fases de análisis, compilación y ejecución están fusionadas en un sistema monolítico, imposibilitando optimizaciones como JIT [5]. Como consecuencia, extender la funcionalidad del *pattern matcher* o agregar optimizaciones es extremadamente complicado y difícil de realizar *debugging*. El escenario deseado es una arquitectura modular con *front-end* (compilador a IR), *middle-end* (optimizaciones) y *backend* (VM). Como contraste, los *frameworks* como LLVM muestran beneficios al tener sistemas modulares [6].

Adicionalmente, el *pattern matcher* del WL realiza asignaciones de memoria frecuentes para objetos temporales, generando computo innecesario en el manejo de memoria automática [7] y dañan la localidad del cache de memoria, especialmente visible en funciones como ReplaceRepeated.

### Problema seleccionado

Este proyecto aborda la falta de un sistema de ejecución eficiente y modular para el *pattern matcher* del Wolfram Language, que resuelva:

1. Límites de profundidad por recursión
2. Falta de optimizaciones debido a acoplamiento del sistema
3. Ineficiencia en gestión de memoria

El problema se puede resumir de la siguiente manera:

El *pattern matcher* de Wolfram Language está fundamentalmente limitado por su diseño monolítico, uso de algoritmo recursivo, y su gestión de memoria ad-hoc, lo que lo vuelve lento, inflexible e inadecuado para la optimización usando técnicas modernas

## Objetivos

### Objetivo general

Diseñar una máquina virtual (VM) como propuesta para modernizar el sistema de *pattern matching* del *Wolfram Language*, reemplazando su implementación recursiva actual por un modelo basado en *bytecode* [8] iterativo, resolviendo así los problemas de límites de profundidad (*stack overflow*) y tiempos de ejecución lentos, con gestión de memoria secundaria. La arquitectura propuesta separa claramente la compilación de patrones (*front-end* que genera *bytecode* optimizable) de su ejecución (*back-end* en la VM), manteniendo compatibilidad exacta con la semántica original mientras habilita extensiones futuras como compilación JIT o paralelismo.

### Objetivos específicos

OE1: Implementar un sistema de ejecución basado en instrucciones lineales

Transformar el enfoque recursivo actual en un modelo de ejecución secuencial que procese patrones mediante una serie de operaciones básicas. Esto implica desarrollar una máquina virtual que procese instrucciones especializadas (en forma de *bytecode*) capaz de representar todas las variantes de patrones del Wolfram Language (desde patrones simples como \_ hasta estructuras complejas con anidamiento y condiciones). El sistema debe manejar el flujo de control mediante mecanismos de salto explícitos que reemplacen las llamadas recursivas, permitiendo así procesar patrones arbitrariamente profundos sin riesgo de desbordamiento de pila. Un componente crítico será el diseño de un esquema de registros para almacenar capturas de patrones (como x\_) que preserve el ámbito léxico exacto del sistema actual.

OE2: Estructurar el sistema de *pattern matching* en fases independientes

Dividir el sistema monolítico actual en módulos especializados con responsabilidades claramente delimitadas. La primera fase se encargará de analizar los patrones y convertirlos a una representación intermedia (IR) [9] estructurada que preserve su semántica. Una segunda fase aplicará transformaciones para optimizar esta representación (como combinar operaciones redundantes o precalcular verificaciones). La fase final traducirá esta representación optimizada a instrucciones ejecutables (*bytecode*), con la posibilidad de generar diferentes variantes según el contexto (como versiones especializadas para patrones frecuentes). Esta separación permitirá desarrollar y optimizar cada componente de manera independiente.

## Resultados Esperados

Tabla 1. Tabla de resultado esperados del proyecto de tesis.

|  |  |  |
| --- | --- | --- |
| **Objetivo Específico 1:**  Diseñar un sistema de ejecución basado en instrucciones lineales que como alternativa al enfoque recursivo actual del *pattern matcher* del Wolfram Language. | | |
| **Resultado Esperado** | **Indicador objetivamente verificable** | **Medio de verificación** |
| Diseño y especificación del *Instruction Set Architecture* (ISA) | * Conjunto completo de instrucciones básicas (MATCH\_HEAD, MATCH\_BIND, etc.) documentadas * Cobertura total para patrones atómicos (\_, x\_, f[x\_], etc.) | * Documento técnico del ISA, especificando cada instrucción (i.e. nombre, argumentos, etc.) * Definición de representación textual del ISA |
| Diseño y especificación del compilador de patrones a instrucciones (*bytecode*) funcional | * Desarrollar la semántica de transformación de patrones a la representación intermedia (IR), y posteriormente a las instrucciones definidas en el ISA | * Documento técnico que especifica conversión de la representación intermedia a las instrucciones del ISA * Definición de representación textual de la representación intermedia (IR) |
| Diseño y especificación de motor de ejecución de las instrucciones | * Desarrollar la semántica de ejecución de las instrucciones generadas previamente | * Documento técnico del ISA, especificando la semántica de cada instrucción (y sus paralelos a las funciones de patrones que existen en WL) |

|  |  |  |
| --- | --- | --- |
| **Objetivo Específico 2:**  Estructurar e implementar el sistema de *pattern matching* en fases independientes. | | |
| **Resultado Esperado** | **Indicador objetivamente verificable** | **Medio de verificación** |
| Implementar un *front-end* para la máquina virtual (*parser*) | * Implementar un *parser* que convierta patrones de *Wolfram* *Language* a una representación intermedia (IR) (diseñado en el objetivo 1.1) * Implementar un API que reciba el patrón y retorne la representación intermedia, p. ej. PatternToIR[patt\_] | * Pruebas unitarias * Verificación de soporte para *scoping* de símbolos (Module, With, Block) * Representación textual de la representación intermedia * Sistema de rastreo de pasos (*tracing*) de compilación con métricas |
| Diseñar e implementar un *middle-end* para la máquina virtual (optimizador) | * Implementar un optimizador que trabaje sobre la representación intermedia (IR) y aplique optimizaciones clave tales como la eliminación de redundancias y especialización de patrones * Implementar un generador de *bytecode* optimizado utilizando el IR como input (diseñado en el objetivo 1.2), p. ej. IRToBytecode[ir\_] | * Pruebas unitarias * Comparación entre semántica de ejecución con y sin optimizaciones * Estadísticas de reducción de tamaño del IR * *Logging* de optimizaciones |
| Diseñar e implementar un *back-end* para la máquina virtual (*backend*) | * Implementar un motor de ejecución iterativo del *bytecode* generado, especificado en el objetivo 1.3 * Sistema para registro de capturas de símbolos (*binding*), p. ej. x\_ guardado en el registro R1 * Implementar un API que reciba *bytecode* y ejecute el *matching* con expresiones de entrada, p. ej. PatternMatchQ[vm, expr\_] | * Pruebas unitarias. En particular, que prueben que la semántica del nuevo *pattern matcher* queda inalterada * Métricas de tiempo de ejecución * *Tracing* de la ejecución de la máquina virtual (p. ej. monitoreo de los contenidos de los registros durante la ejecución) |

## Métodos y Procedimientos

A continuación, se presenta una tabla que relaciona los resultados esperados con los métodos y procedimientos propuestos para alcanzarlos. Cada método se describe brevemente, junto con las herramientas o técnicas que se utilizarán y su fundamentación teórica.

Para garantizar el cumplimiento de los resultados esperados, se han definido los siguientes métodos y procedimientos:

|  |  |  |  |
| --- | --- | --- | --- |
| **Resultado Esperado** | **Método** | **Descripción** | **Herramientas/Técnicas** |
| Diseño del ISA | *Instruction Set Design* | Definición de instrucciones básicas basadas en patrones WL. Análisis de casos comunes para cubrir semántica existente. | * Análisis estático de patrones WL. * Benchmarking de operaciones frecuentes. * WASM como referencia de diseño. |
| Compilador de patrones a bytecode | *Parsing* y generación de IR | Conversión de patrones WL a una IR estructurada mediante *parsing* recursivo-descendente. Validación de ámbito léxico. | * *Parser* en Wolfram Language. * Representación gráfica del AST. * Pruebas de compatibilidad. |
| Motor de ejecución iterativo | Máquina virtual con registros fijos | Implementación de un loop *fetch-decode-execute* con:   * Saltos explícitos (reemplazo de recursión). * Registros pre-asignados para capturas. | * VM en Wolfram Language (prototipo). * *Tracing* de ejecución. * WASM como modelo de referencia. |
| Optimizador de IR | Eliminación de redundancias | Transformación de la IR para:   * Combinar verificaciones redundantes. * Precalcular patrones constantes. * Especializar patrones frecuentes. | * Manipulación simbólica del IR. * *Logging* de optimizaciones aplicadas. |
| Validación semántica | Pruebas de equivalencia | Generación de casos de prueba para verificar que la VM reproduce la semántica del sistema actual (p. ej. MatchQ, ReplaceAll). | * Wolfram Testing Framework. * Fuzzing con patrones aleatorios. * Comparación de resultados con implementación nativa. |
| Benchmarking | Comparación de rendimiento | Medición de:   * Tiempos de ejecución (patrones profundos vs. sistema actual). * Profundidad máxima soportada (sin stack overflow). | * Perfilamiento con AbsoluteTiming. * *Datasets* de patrones estandarizados. |

Herramientas clave:

* Wolfram Language: Para implementación del *parser*, generación de IR y pruebas.
* *WASM* [10] (como referencia): Inspiración para el diseño modular del pipeline (front-end/middle-end/back-end), y la especificación técnica de la máquina virtual.
* *Benchmarking*: Mediciones de rendimiento vs. implementación actual (MatchQ, ReplaceAll).
* *Tracing*: Instrumentación de la VM para depuración (p. ej., visualización de ejecución paso a paso).

## Justificación

El proyecto aborda limitaciones críticas del sistema actual de pattern matching en Wolfram Language, que impactan su escalabilidad y rendimiento. La recursión no optimizada restringe el manejo de patrones complejos, mientras que la arquitectura monolítica dificulta la incorporación de optimizaciones modernas (p. ej. JIT). Aunque existen soluciones para problemas similares en otros dominios (p. ej. regex o Prolog), no se han adaptado al pattern matching simbólico de WL, creando una brecha en la literatura. La propuesta de una VM especializada no solo resolvería estos problemas técnicos (como el stack overflow y la gestión de memoria), sino que también sentaría las bases para futuras extensiones (paralelismo, compilación JIT). El impacto práctico es significativo: operaciones como ReplaceAll o Cases son fundamentales en WL, y su optimización beneficiaría a usuarios en áreas como procesamiento simbólico, manipulación de datos y desarrollo de algoritmos.

## Delimitación y Alcance

El proyecto se enfoca en el diseño e implementación de una VM para *pattern matching*, excluyendo modificaciones al lenguaje WL o a su sintaxis. Se priorizarán patrones estándar (e.g., \_, x\_, f[x\_]), dejando para trabajo futuro patrones avanzados (p. ej. dependientes de tipos o condiciones complejas). La VM se implementará como un prototipo en WL, sin integración al Wolfram Language. El prototipo final estará implementado usando un lenguaje compilado.

Las métricas de éxito incluirán:

* Compatibilidad semántica: Equivalencia funcional con el sistema actual.
* Rendimiento: Reducción de tiempo de ejecución en patrones profundos (>1000 niveles de profundidad).
* Memoria: Menor uso de memoria temporal en *benchmarks* estandarizados.

El alcance técnico cubre desde el *parsing* hasta la ejecución, pero no incluye generación de código nativo (JIT) ni paralelismo, aunque se diseñará la arquitectura para permitir estas extensiones.

# Marco Conceptual

## Introducción

## *Wolfram Language* (WL) [2]

Lenguaje de alto nivel y multiparadigma, desarrollado por Wolfram Research [11]. Se especializa en: 1) programación simbólica y funcional, 2) programación multiparadigma y programación basado en reglas, y 3) puede emplear estructuras y datos arbitrarios.

## *Pattern matching* [1]

Mecanismo para identificar y manipular estructuras de datos basado en su forma o patrones definidos. En el caso particular del *Wolfram Language*, los patrones son fundamentales para el desarrollo de sus programas.

## *(Process) Virtual machine* (VM)

En español, máquina virtual de aplicación, es un sistema software que ejecuta código intermedio (*bytecode*) en un entorno independiente del hardware [3].

## *Bytecode*

Representación intermedia compacta de instrucciones ejecutables por una máquina virtual (de aplicación) [8].

## *Instruction Set Architecture* (ISA)

Conjunto de instrucciones y registros definidos para un procesador o máquina virtual [3].

## Compilación

Proceso de traducción de código fuente (en este proyecto, patrones de *Wolfram Language*) a un formato ejecutable (en este proyecto, a instrucciones o *bytecode*) [8].

## *Intermediate Representation* (IR)

Estructura de datos que representa código en una fase intermedia de compilación, facilitando optimizaciones [9].

## *Just-in-Time compilation* (JIT)

Técnica que compila código a máquina nativa durante la ejecución para mejorar rendimiento [5].

## *Static Analysis*

Evaluación de código sin ejecutarlo para detectar errores o posibles optimizaciones [12].

## *Tail-Code Optimization* (TCO)

Optimización que reemplaza llamadas recursivas en cola por iteraciones para evitar desbordamiento de pila [4].

## *Memory Management*

Mecanismo para controlar la asignación y liberación de memoria durante la ejecución [10].

## *Memory Allocation*

Asignación dinámica o estática de espacio en memoria para datos durante la ejecución [7].

## *Memory Pooling*

Técnica de gestión de memoria que reutiliza regiones preasignadas para reducir *overhead* [13].

## *Debugging*

Proceso de identificar y corregir errores en programas mediante herramientas de trazado y análisis [14].

## *Parser*

Un programa que analiza (de forma sintáctica) y convierte un lenguaje de entrada (en este proyecto una expresión de WL) a un lenguaje de salida (en este proyecto a un IR o *bytecode*).

# Revisión del Estado del Arte

## Introducción

El *pattern matching* es un mecanismo fundamental en lenguajes de programación simbólica como Wolfram Language (WL), pero su implementación actual en WL enfrenta limitaciones en escalabilidad y rendimiento. Esta revisión analiza soluciones existentes en tres ejes clave:

1. técnicas de optimización para *pattern matching*
2. diseño de máquinas virtuales (VMs) especializadas
3. estrategias de compilación a *bytecode*

El objetivo es identificar brechas en la literatura y fundamentar la propuesta de una VM para implementar el *pattern* *matching* en WL.

## Objetivos de revisión

1. Identificar enfoques para optimizar implementaciones de *pattern matching* en lenguajes funcionales/simbólicos
2. Analizar arquitecturas de VMs y esquemas de compilación (IR, *bytecode*) aplicables a operaciones simbólicas
3. Evaluar investigaciones previas sobre el rendimiento o propuestas de optimización del *pattern matcher* de WL

## Preguntas de revisión

* P1. ¿Qué técnicas de optimización para *pattern matching* en lenguajes simbólicos o funcionales (p. ej. Haskell, Prolog, Lisp) han demostrado mejorar el rendimiento o escalabilidad?
* P2. ¿Cómo se han abordado el diseño de máquinas virtuales y los esquemas de compilación (IR, *bytecode*, optimizaciones) para sistemas de *pattern matching* u operaciones simbólicas en lenguajes comparables?
* P3. ¿Qué estudios existen sobre las limitaciones del sistema actual de *pattern matching* en Wolfram Language (p. ej. rendimiento, profundidad de recursión), y qué soluciones se han sugerido?

## Estrategia de Búsqueda

### Motores de búsqueda a usar

1. ACM Digital Library (teoría de lenguajes y compiladores)
2. IEEE Xplore (énfasis en ingeniería de software y VMs)

### Cadenas de búsqueda a usar

1. ("pattern matching" OR “pattern matcher”) AND ("optimization" OR "design" OR “implementation” OR “techniques”) AND (“functional” OR “symbolic”)
2. ("virtual machine" OR "bytecode") AND (“pattern matcher” OR “pattern matching” OR "symbolic" OR "rule-based systems")
3. (“Wolfram Language” OR “Mathematica”) AND (“pattern matcher” OR “pattern matching”)

### Documentos encontrados

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pregunta** | **Motor de Búsqueda** | **Title** | **Link** | **Año** |
| 1, 2 | ACM | Optimising First-Class Pattern Matching | <https://doi.org/10.5281/zenodo.7189905> | 2022 |
| 1, 2 | ACM | [Embedded Pattern Matching](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-2'!A1) | <https://doi.org/10.1145/3554299> | 2022 |
| 1, 2 | ACM | [Rigorous Pattern Matching as a Language Feature](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-3'!A1) | <https://doi.org/10.1145/3591335.3591342> | 2023 |
| 1, 2 | ACM | [Pattern Matching in AI Compilers and Its Formalization](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-4'!A1) | <https://doi.org/10.1145/3696443.3708934> | 2025 |
| 1, 2 | ACM | [Dynamic pattern matching with Python](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-5'!A1) | <https://doi.org/10.1145/3426422.3426983> | 2020 |
| 1, 2 | ACM | [Compiling pattern matching to in-place modifications](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-6'!A1) | <https://doi.org/10.1145/3486609.3487204> | 2021 |
| 1 | IEEE | Design of high performance pattern matching engine through compact deterministic finite automata | <https://doi.org/10.1145/1391469.1391685> | 2008 |
| 1, 2 | ACM | [Efficient Pattern Matching in Python](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-8'!A1) | <https://doi.org/10.1145/3149869.3149871> | 2017 |
| 1, 2 | ACM | [Optimizing pattern matching](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-9'!A1) | <https://doi.org/10.1145/507635.507641> | 2001 |
| 1, 2 | ACM | [Open pattern matching for C++](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-10'!A1) | <https://doi.org/10.1145/2508075.2508098> | 2013 |
| 1, 2 | ACM | [Compiling pattern matching](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-11'!A1) | <https://dl.acm.org/doi/10.5555/5280.5303> | 1985 |
| 1, 2 | ACM | [Compiling lazy pattern matching](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-12'!A1) | <https://doi.org/10.1145/141471.141499> | 1992 |
| 1, 2 | ACM | [Compiling pattern matching to good decision trees](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-13'!A1) | <https://doi.org/10.1145/1411304.1411311> | 2008 |
| 1, 2 | ACM | [Left-to-Right Tree Pattern Matching](applewebdata://E37972FA-A93B-45E4-8878-E2159D516E1D#'Extracci%C3%B3n-14'!A1) | <https://dl.acm.org/doi/abs/10.5555/647192.720508> | 1991 |

## Formulario de extracción de datos

|  |  |  |  |
| --- | --- | --- | --- |
| **Pregunta** | **Campo** | **Descripción** | **Ejemplo** |
| **1** | Técnica | Nombre y tipo de optimización | "Compilación a árboles de decisión |
| 1 | Lenguaje | Lenguaje de implementación | "Haskell", "Prolog |
| 1 | Beneficios | Ventajas cuantificables | "40% faster for nested patterns" |
| 1 | Limitaciones | Restricciones o casos no soportados | "No soporta pattern guards" |
| 1 | Adaptación a WL | Factibilidad para Wolfram Language | "Aplicable a ReplaceAll" |
| **2** | Arquitectura VM | Diseño base de la VM | "Register-based with 16 slots" |
| 2 | IR | Tipo de representación intermedia | "SSA Form", "Control Flow Graph" |
| 2 | Optimizaciones | Técnicas aplicadas al bytecode | "Dead code elimination", "Peephole opt." |
| 2 | Modelo Ejecución | Estrategia de implementación | "JIT with fallback interpreter" |
| 2 | Manejo Memoria | Sistema de gestión de memoria | "Arena allocation for pattern captures" |
| **3** | Críticas | Problemas identificados en WL | "Stack overflow en patrones >1000 niveles" |
| 3 | Soluciones Propuestas | Mejoras sugeridas (si existen) | "Use memoization for repeated patterns" |
| 3 | Brechas | Áreas no cubiertas por estudios previos | "No solución para memory leaks" |

## Resultados de la revisión

### Respuesta a pregunta P1: Técnicas de optimización para lenguajes simbólicos/funcionales

La revisión de literatura muestra que las técnicas de optimización más efectivas para *pattern* *matching* en lenguajes funcionales y simbólicos caen en tres categorías principales. En primer lugar, la compilación a árboles de decisión con *maximal sharing* [15], implementada en lenguajes como OCaml y Haskell, ha demostrado reducir el uso de memoria hasta en un 60% para patrones anidados mediante la eliminación de redundancias en las estructuras de *matching*. Esta técnica es particularmente relevante para el Wolfram Language, ya que podría optimizar operaciones fundamentales como ReplaceAll y el manejo de patrones condicionales complejos [16][17].

Un segundo enfoque destacado es el uso de autómatas deterministas (DFA) para la verificación estática de patrones, empleado en lenguajes como Ada y ParaSail. Este método no solo garantiza la exhaustividad y no ambigüedad de los patrones [18], sino que también permite saltos directos a las acciones correspondientes, mejorando significativamente el rendimiento. Para el Wolfram Language, esta técnica podría aplicarse para optimizar la construcción de estructuras como DownValues y verificar patrones durante la fase de compilación [19].

Finalmente, técnicas avanzadas como el *matching in-place* con *scheduling* poliédrico [20], implementado en lenguajes similares a Lisp como Rew, ofrecen mejoras sustanciales en el manejo de memoria para patrones profundos. Aunque estos trabajos no abordan específicamente los patrones con condiciones dinámicas (/;) característicos del Wolfram Language [21], sientan bases técnicas importantes.

### Respuesta a pregunta P2: Diseño de VMs y esquemas de compilación

El análisis de los esquemas de compilación y diseño de máquinas virtuales para *pattern* *matching* revela varios modelos arquitectónicos relevantes. Las implementaciones más exitosas, como las de OCaml y otros lenguajes funcionales [15][22], se basan en máquinas virtuales *stack-based* con bytecode especializado que incluye instrucciones como SWITCH para bifurcaciones y PUSH\_CAPTURE para el manejo de variables de patrón. Este enfoque ha demostrado ser particularmente efectivo para eliminar los problemas de *stack overflow* asociados a implementaciones recursivas [23], al tiempo que proporciona un mecanismo eficiente para el manejo de capturas de patrones.

Por otro lado, sistemas más modernos como los empleados en DSLs de Python [24] utilizan representaciones intermedias modulares basadas en grafos de operaciones. Esta arquitectura permite una clara separación entre las fases de análisis, optimización y ejecución, facilitando la implementación de técnicas avanzadas como compilación JIT y paralelismo. Para el Wolfram Language, esta separación de responsabilidades sería crucial para superar las limitaciones del sistema monolítico actual [25].

Un tercer modelo relevante es la compilación en etapas (*staged compilation*) empleada en Haskell [26] para el manejo de *pattern synonyms*. Este enfoque genera código especializado adaptado a contextos específicos, ofreciendo interesantes posibilidades para optimizar operaciones frecuentes en el Wolfram Language como en ReplaceAll. La combinación de estos modelos - bytecode lineal, IR optimizable y compilación especializada - constituye la base más sólida para el diseño propuesto en esta tesis.

### Respuesta a pregunta P3: Limitaciones del *pattern* *matcher* en WL

Aunque la literatura no aborda específicamente el sistema de *pattern matching* del Wolfram Language, identifica problemas análogos en otros lenguajes que comparten características similares. Los sistemas basados en implementaciones recursivas, como se encuentra en ML tradicional, enfrentan idénticos límites de profundidad que conducen a *stack overflow*, problema que ha sido resuelto satisfactoriamente mediante la transición a modelos basados en bytecode iterativo.

De igual manera, la falta de optimizaciones en sistemas acoplados como Python refleja las limitaciones actuales del Wolfram Language, donde la fusión de fases de compilación y ejecución impide aplicar transformaciones significativas. La solución validada en estos contextos - una arquitectura modular con representaciones intermedias optimizables - ofrece un camino claro para superar estas restricciones.

En cuanto al manejo de memoria, los problemas reportados en Haskell con asignaciones temporales para capturas de patrones son notablemente similares a los observados en el Wolfram Language. Las soluciones propuestas en la literatura, que incluyen el uso de registros y layouts de memoria planos, proporcionan técnicas comprobadas que podrían adaptarse al contexto específico del Wolfram Language. Estos antecedentes respaldan técnicamente las soluciones propuestas en esta tesis, mientras que la originalidad radica en su adaptación a las características únicas del sistema de *pattern matching* del Wolfram Language.

## Conclusiones

El análisis sistemático de la literatura revela que las técnicas más promisorias para modernizar el sistema de *pattern matching* del Wolfram Language son la compilación a bytecode con *maximal sharing* y el uso de representaciones intermedias modulares. Estas aproximaciones han demostrado su eficacia para resolver problemas idénticos en otros lenguajes funcionales y simbólicos, particularmente en lo que respecta a la eliminación de recursión, la optimización de patrones complejos y la mejora en el manejo de memoria.

La arquitectura propuesta en esta tesis sintetiza los mejores elementos de los modelos existentes, combinando la eficiencia de las máquinas virtuales (*stack-based* o *register-based*, aún por decidir) con la flexibilidad de los sistemas basados en grafos de operaciones. Esta combinación no solo aborda las limitaciones actuales del Wolfram Language, sino que además establece las bases para futuras extensiones como la compilación JIT y el paralelismo.

La principal contribución de este trabajo radica en la adaptación de estas técnicas al dominio específico del Wolfram Language, particularmente en el manejo de patrones con condiciones dinámicas y variables simbólicas, aspectos no cubiertos en profundidad por la literatura existente/revisada. Para la implementación práctica, se recomendaría priorizar el desarrollo de mecanismos de *maximal sharing* en la representación intermedia (IR), un sistema de registros con ámbito léxico para capturas de patrones, y un subsistema de generación de perfiles que permita optimizaciones adaptativas basadas en patrones de uso frecuente (probablemente esto sea un trabajo a futuro).

# Referencias

[1] “Patterns—Wolfram Language Documentation.” Accessed: Apr. 05, 2025. [Online]. Available: https://reference.wolfram.com/language/tutorial/Patterns.html#139

[2] “Wolfram Language: Programming Language + Built-In Knowledge.” Accessed: Apr. 21, 2025. [Online]. Available: https://www.wolfram.com/language/

[3] J. E. Smith and Ravi Nair, “The architecture of virtual machines,” *Computer*, vol. 38, no. 5, pp. 32–38, May 2005, doi: 10.1109/MC.2005.173.

[4] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[5] J. Aycock, “A brief history of just-in-time,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003, doi: 10.1145/857076.857077.

[6] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, Mar. 2004.

[7] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, “Dynamic storage allocation: A survey and critical review,” in *Memory Management*, vol. 986, H. G. Baler, Ed., in Lecture Notes in Computer Science, vol. 986. , Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. doi: 10.1007/3-540-60368-9\_19.

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, “The Implementation of Lua 5.0”.

[9] “LLVM Bitcode File Format — LLVM 21.0.0git documentation.” Accessed: Apr. 21, 2025. [Online]. Available: https://llvm.org/docs/BitCodeFormat.html

[10] “WebAssembly Specification — WebAssembly 2.0 (Draft 2025-01-28).” Accessed: Apr. 21, 2025. [Online]. Available: https://webassembly.github.io/spec/core/

[11] “Wolfram Research Company Background.” Accessed: Apr. 21, 2025. [Online]. Available: https://www.wolfram.com/company/

[12] B. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and W. Marsh, “Industrial perspective on static analysis,” *Softw. Eng. J.*, vol. 10, pp. 69–75, Apr. 1995, doi: 10.1049/sej.1995.0010.

[13] J. Barker, *Beginning Java Objects: From Concepts to Code*. 2005. doi: 10.1007/978-1-4302-0036-9.

[14] “What Is Debugging? | IBM.” Accessed: Apr. 21, 2025. [Online]. Available: https://www.ibm.com/think/topics/debugging

[15] L. Maranget, “Compiling pattern matching to good decision trees,” in *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, Victoria BC Canada: ACM, Sep. 2008, pp. 35–46. doi: 10.1145/1411304.1411311.

[16] J. Smits, View Profile, T. Hartman, View Profile, J. Cockx, and View Profile, “Optimising First-Class Pattern Matching,” *Proc. 15th ACM SIGPLAN Int. Conf. Softw. Lang. Eng.*, pp. 74–83, Nov. 2022, doi: 10.1145/3567512.3567519.

[17] A. Gräf, “Left-to-right tree pattern matching,” in *Rewriting Techniques and Applications*, R. V. Book, Ed., Berlin, Heidelberg: Springer, 1991, pp. 323–334. doi: 10.1007/3-540-53904-2\_107.

[18] S. T. Taft and View Profile, “Rigorous Pattern Matching as a Language Feature,” *ACM SIGAda Ada Lett.*, vol. 42, no. 2, pp. 69–74, Apr. 2023, doi: 10.1145/3591335.3591342.

[19] J. W. Cutler *et al.*, “Pattern Matching in AI Compilers and Its Formalization,” *Proc. 23rd ACMIEEE Int. Symp. Code Gener. Optim.*, pp. 63–76, Mar. 2025, doi: 10.1145/3696443.3708934.

[20] P. Iannetta *et al.*, “Compiling pattern matching to in-place modifications,” *Proc. 20th ACM SIGPLAN Int. Conf. Gener. Program. Concepts Exp.*, pp. 123–129, Oct. 2021, doi: 10.1145/3486609.3487204.

[21] T. Kohn *et al.*, “Dynamic pattern matching with Python,” *Proc. 16th ACM SIGPLAN Int. Symp. Dyn. Lang.*, pp. 85–98, Nov. 2020, doi: 10.1145/3426422.3426983.

[22] L. Maranget, “Compiling lazy pattern matching,” in *Proceedings of the 1992 ACM conference on LISP and functional programming*, in LFP ’92. New York, NY, USA: Association for Computing Machinery, Jan. 1992, pp. 21–31. doi: 10.1145/141471.141499.

[23] F. Le Fessant and L. Maranget, “Optimizing pattern matching,” in *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, in ICFP ’01. New York, NY, USA: Association for Computing Machinery, Oct. 2001, pp. 26–37. doi: 10.1145/507635.507641.

[24] M. Krebber, View Profile, H. Barthels, View Profile, P. Bientinesi, and View Profile, “Efficient Pattern Matching in Python,” *Proc. 7th Workshop Python High-Perform. Sci. Comput.*, pp. 1–9, Nov. 2017, doi: 10.1145/3149869.3149871.

[25] Y. Solodkyy, View Profile, G. Dos Reis, View Profile, B. Stroustrup, and View Profile, “Open pattern matching for C++,” *Proc. 2013 Companion Publ. Conf. Syst. Program. Appl. Softw. Humanity*, pp. 97–98, Oct. 2013, doi: 10.1145/2508075.2508098.

[26] T. L. McDonell, J. D. Meredith, and G. Keller, “Embedded pattern matching,” in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, in Haskell 2022. New York, NY, USA: Association for Computing Machinery, Sep. 2022, pp. 123–136. doi: 10.1145/3546189.3549917.