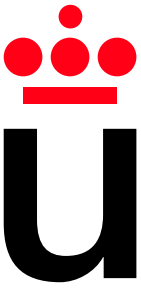

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2024–2025

Trabajo Fin de Grado

Implementación de un algoritmo para optimizar la evaluación de *Expression Templates*

Autor: Sergio Zarco Mármol
Tutor: José Ángel Moreno San Segundo

18 de junio de 2025

© 2025 Sergio Zarco Mármol
Algunos derechos reservados
Este documento se distribuye bajo la licencia Atribución 4.0 Internacional de Creative Commons,
disponible en: https://creativecommons.org/licenses/by/4.0/deed.es

Resumen

Este trabajo presenta una mejora en la evaluación de expresiones representadas mediante *Expression Templates*, una técnica ampliamente utilizada en sistemas de Computación de Altas Prestaciones para evitar la creación innecesaria de objetos temporales y optimizar operaciones complejas en tiempo de compilación

La propuesta introduce un enfoque alternativo basado en la metaprogramación y en técnicas de procesamiento de lenguajes, para transformar el árbol que representa la expresión en un grafo que optimiza su estructura, mejorando el rendimiento en tiempo de ejecución.
Para ello, previo a la evaluación de la expresión, se construye una estructura que elimina las subexpresiones repetidas, reduciendo la cantidad de cómputo necesario para evaluarla, mejorando así su rendimiento.

Los resultados experimentales muestran mejoras en el rendimiento hasta 13 veces mejor que implementaciones tradicionales, demostrando el potencial del proyecto y la importancia de la búsqueda constante de la mejora de rendimiento en el entorno de la Computación de Altas Prestaciones.

Palabras clave:

- Expression Templates
- Computational Fluid Dynamics (CFD)
- Metaprogramación
- Computación de Altas Prestaciones (HPC)

Índice de contenidos

Índice de tablas **VI**

Índice de figuras **VII**

Índice de códigos **VIII**

1. Introducción **1**

2. Objetivos **7**

3. Estado del arte **9**

4. Tecnologías, Herramientas y Metodologías **13**
4.1. Tecnologías y Herramientas 13
4.2. Metodología 15

5. Descripción Informática **17**
5.1. Requisitos 17
5.2. Arquitectura y Análisis 18
5.3. Diseño e Implementación 18
5.3.1. Sustitución de expresiones constantes 18
5.3.2. Algoritmo 20
5.3.3. Construcción del Grafo en tiempo de compilación 22
5.3.4. Implementación basada en Polimorfismo Dinámico 23
5.3.5. Implementación basada en polimorfismo estático 28

6. Análisis de rendimiento **35**
6.1. Hardware empleado 35
6.2. Sistema de medición de tiempos 36
6.3. Programa de pruebas 36
6.4. Resultados obtenidos 37
6.4.1. Expresiones Simples 37
6.4.2. Polimorfismo Dinámico 38
6.4.3. Polimorfismo Estático 40

6.4.4. Caso de Uso Realista 41

7. Conclusiones y trabajos futuros **43**

7.1. Mejoras a futuro 44

Bibliografía **46**

Índice de tablas

6.1. Tiempos Simplificación expresiones simples 38
6.2. Tiempos Polimorfismo Dinámico 39
6.3. Tiempos Polimorfismo Estático 40

Índice de figuras

1.1. simulación *CFD* de un coche. 1

5.1. Árbol correspondiente a la expresión superior. 21
5.2. GDA Correspondiente a la expresión superior. 22
5.3. Diagrama de clases utilizadas en la construcción del grafo. 24
5.4. Diagrama de clases haciendo uso de *SFINAE* 29

6.1. Mejora de rendimiento relativa al uso de temporales 41

Índice de códigos

5.1. struct *is_1D* 19
5.2. Declaración *motherclass* en *binaryExpression* 19
5.3. Declaración de *binaryExpression* con *is_1D* 20
5.4. *operator==* de *ADGElement* para *binaryExpression* 25
5.5. *operator[]* de *ADGElement* para *binaryExpression* 25
5.6. Construcción del grafo para *binaryExpression* 26
5.7. Especialización de *ADGElement* para *binaryExpression* 30
5.8. Construcción del grafo para *binaryExpression* con polimorfismo estático 32
5.9. Función *assignFieldFactorized* 33

1 Introducción

El campo de la dinámica de fluidos computacional, *CFD* por sus siglas en inglés, surgió en la década de 1960, cuando se publicó el primer artículo que incluía un modelo tridimenisional, del avión *Douglas*[1]. Con el paso de los años, más equipos de investigación comenzaron a utilizar computadores para simular modelos tridimensionales en investigación, principalmente aplicándolos en aviones y barcos.
Conforme la capacidad de los computadores ha aumentado, también lo han hecho los resultados que puede ofrecer este campo. Esto ha permitido que ya no solo investigadores hagan uso de estas tecnologías. Hoy en día, la simulación de dinámica de fluidos tiene aplicaciones en multitud de áreas, como diseño de vehículos, estudio del clima[2], industria alimentaria[3], previsión meteorológica, etc.

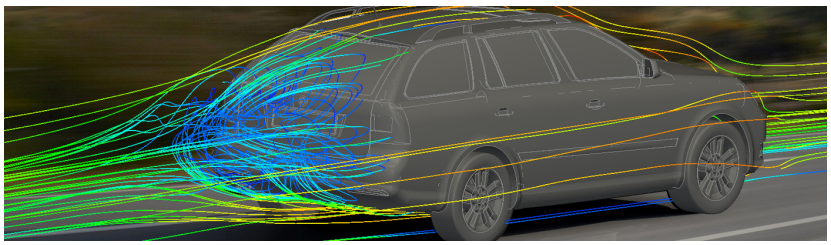

Figura 1.1: simulación *CFD* de un coche.

Una de las principales fortalezas de la dinámica de fluidos computacional radica en su capacidad para resolver las ecuaciones que rigen los fenómenos físicos en su forma diferencial. Esta aproximación rigurosa permite describir con alta fidelidad el comportamiento de sistemas complejos bajo condiciones bien definidas, basándose en principios fundamentales de la física.

Gracias a esto, los programas *CFD* poseen una capacidad predictiva intrínseca, es decir, pueden anticipar el comportamiento de un sistema incluso en escenarios no observados previamente, siempre que las condiciones del modelo estén correctamente planteadas. Esta capacidad de extrapolación representa una ventaja significativa respecto a otras aproximaciones basadas en técnicas empíricas o en inteligencia artificial, las cuales, si bien pueden ser muy eficaces dentro del rango de datos con los que han sido entrenadas, muestran limitaciones cuando se enfrentan a situaciones fuera de dicho dominio.

Uno de los desafíos más importantes a los que se enfrenta este tipo de programas es el rendimiento que ofrecen, con simulaciones que pueden durar días[4]. Esto se debe en gran parte a su funcionamiento basado en operaciones que se realizan sobre grandes conjuntos de datos, y se repiten millones de veces.

Por esta razón, resulta crucial la implementación de optimizaciones por leves que sean, ya que pueden resultar en mejoras importantes en el rendimiento global del programa. La importancia de las optimizaciones en este ámbito se ve directamente reflejada en la cantidad de artículos científicos que se publican anualmente sobre este ámbito, siendo en el año 2024 más del doble que la tendencia en años anteriores.

Para lograr obtener el máximo rendimiento posible, originalmente se utilizaban lenguajes de bajo nivel como *Fortran*[5], que proporcionan el máximo rendimiento. Sin embargo, utilizar estos lenguajes en la actualidad supone grandes problemas. Los programas de *CFD* están diseñados para resolver gran cantidad de problemas diferentes, que, aunque comparten una base común, difieren en algunos aspectos. Los simuladores permiten incluir gran cantidad de configuraciones al usuario final (condiciones de contorno, divergencias, etc.), a las que se debe adaptar el código del programa.

Esto supone que el programa debe poder adaptarse a muchas situaciones diferentes, lo cual resulta complicado de implementar en lenguajes de bajo nivel, que carecen de herramientas como orientación a objetos, polimorfismo, etc. Es por ello que estos lenguajes se suelen descartar en aplicaciones de gran calibre, que requerirían códigos muy extensos, lo que comprometería su mantenibilidad.

En un intento por combinar el rendimiento de los lenguajes de bajo nivel con las herramientas de alto nivel, muchos programas disponibles en el mercado actualmente optan por utilizar el lenguaje C++. Aunque este lenguaje ofrece un rendimiento menor a los lenguajes de bajo nivel, supera a muchos otros en eficiencia, y, al mismo tiempo, proporciona herramientas que facilitan el desarrollo a corto y largo plazo.

Una de las principales ventajas de C++ es que ofrece un conjunto de herramientas avanzadas que permiten trasladar cómputo y decisiones de tiempo de ejecución a tiempo de compilación. Además de reducir sobrecarga en tiempo de

ejecución, estas herramientas abren la puerta a optimizaciones más agresivas por parte del compilador.

Uno de los pilares de este paradigma es el sistema de plantillas, que permite implementar genéricos sin sacrificar rendimiento. A diferencia de otros lenguajes donde se requieren indirecciones y estructuras de datos dinámicas, en C++ se resuelven en tiempo de compilación, generando código específico para cada tipo involucrado.
Este sistema de generación automática se puede perfeccionar mediante la especialización de plantillas, una técnica que permite definir manualmente implementaciones para tipos particulares con el fin de explotar propiedades específicas de los tipos. Así, se pueden diseñar versiones optimizadas de algoritmos o estructuras de datos cuando se conocen las condiciones, como la clase *vector* del estándar[6].

A partir del sistema de plantillas surge la metaprogramación, que permite escribir programas (o partes de ellos) que se ejecutan íntegramente en tiempo de compilación. Adicionalmente, en C++ moderno se ha potenciado su uso con herramientas como *constexpr* y *consteval*, que facilitan la implementación y mejoran la expresividad.

Otro ámbito en el que destaca este lenguaje es la implementación del polimorfismo, ya que en algunos casos se puede prescindir del polimorfismo dinámico. En ciertas situaciones, si se puede resolver el polimorfismo en tiempo de compilación, se puede optar por utilizar el polimorfismo estático, implementado mediante el patrón *CRTP (Curiously Recurring Template Pattern)*. Esta técnica permite resolver el polimorfismo en tiempo de compilación, eliminando las tablas virtuales y permitiendo el código *inline*, logrando importantes reducciones en el tiempo de ejecución. Sin embargo, su sintaxis es poco natural y entender el flujo de la herencia y la resolución de métodos puede resultar complejo, especialmente para aquellos que no están familiarizados con los patrones avanzados del lenguaje.

Una de las técnicas más potentes, y complejas, son las *Expression Templates*. Esta técnica permite representar operaciones compuestas, como sumas y productos de vectores y matrices, como estructuras construidas en tiempo de compilación. De este modo se logra obtener evaluación perezosa (una operación no se computa hasta ser imprescindible) y muy optimizable por el compilador, evitando la creación de valores intermedios temporales, que en una implementación tradicional serían inevitables, suponiendo una sobrecarga computacional.
En una implementación de sobrecarga de operadores tradicional, expresiones compuestas (a = b + c + d), cada operación intermedia calcula y genera un resultado intermedio, lo cual resulta súmamente ineficiente cuando se trabaja sobre estructuras de datos grandes.
Expression Templates resuelve completamente este problema creado una representación de la expresión como una estructura recursiva de tipo árbol en la que cada nodo representa una operación intermedia. De esta forma, se retrasa la evaluación hasta el momento más conveniente, resolviendo la asignación final como

una operación recursiva entre los nodos del árbol, elemento a elemento, sin necesidad de temporales.

El mecanismo se basa en una jerarquía de clases que representan cada tipo de expresión. Los operadores sobrecargados se modifican para que, en lugar de devolver el resultado de una operación, devuelvan la expresión que representa esa operación. Estos operadores no calculan el resultado, sino que delegan esta tarea a otro momento, típicamente el operador de asignación.
Entre las ventajas más notables del uso de esta técnica se encuentra la eliminación sistemática de los temporales, reduciendo considerablemente el consumo de memoria y tiempo de ejecución. Adicionalmente, representar la expresión como una estructura única permite que el compilador aplique otras técnicas de optimización no aplicables en otros casos, como vectorización de las operaciones, y fusión de bucles.

Todas estas técnicas, junto con muchas otras, aparecen integradas dentro de los programas como una parte natural de su núcleo de cómputo. Esto provoca que el código resulte largo y difícil de comprender, especialmente si se pretende modificar el funcionamiento interno. Sin embargo, la ganancia que se obtiene a cambio en forma de tiempos de simulación menores provoca que, a pesar de las dificultades que implica mantener los códigos, merezca la pena implementar este tipo de optimizaciones.

Las *Expression Templates* en concreto, a pesar de haber sido descubiertas hace varias décadas, siguen sin tener un uso extendido en este tipo de programas, y su conocimiento no es demasiado extenso. Esto es debido, entre otros factores, a que implementarlas en un proyecto ya existente implica hacer una modificación muy agresiva y extensa al núcleo del programa, lo cual puede resultar extremadamente costoso en proyectos grandes, por lo que en proyectos actuales se encuentra en fases de desarollo y testing. Esto hace que sean un campo de estudio con un gran potencial de mejora. Este proyecto se centrará en explotar algunas de sus cualidades para mejorar el rendimiento aún más de lo que se logra en implementaciones actuales[7, 8, 9].

La representación de expresiones como una estructura árbol permite que se puedan implementar optimizaciones a la propia estructura de la expresión utilizando algoritmos, lo cual es imposible si se prescinde del uso de *Expression Templates*.

Este tipo de optimización puede resultar especialmente interesante en expresiones que incluyan una operación compleja más de una vez debido a que, resolver este tipo de expresiones sin utilizar *Expression Templates* obliga al programador a almacenar resultados intermedios, provocando consumos elevados de memoria y tiempo.
Por otra parte, utilizar *Expression Templates* resuelve el problema del consumo de memoria por temporales, que ya no son necesarios, pero sigue resolviendo la

operación repetida tantas veces como aparezca, lo que sigue sin ser ideal.
Puesto que la expresión se representa como una estructura de datos, se puede crear otra estructura a partir del árbol de la expresión que elimine todos los nodos repetidos, resultando en una expresión optimizada que realiza todas las operaciones una única vez, evitando cálculos repetidos.

2 Objetivos

Los objetivos del proyecto son los siguientes:

Algoritmo de optimización
Encontrar un algoritmo que permita optimizar la evaluación de expresiones aritméticas, y adaptarlo para que funcione en el entorno de *Expression Templates*.

Mejora del rendimiento de *Expression Templates*
El objetivo principal de este proyecto es reducir el coste computacional y el tiempo que conlleva evaluar expresiones mediante *Expression Templates*.

Dynamic Dispatching
En caso de ser necesario, se planteará una implementación del algoritmo que no dependa del *Dynamic Dispatching*, ya que este mecanismo de C++ puede ralentizar la ejecución del programa.

Tiempo de Compilación
Un objetivo adicional es lograr que la construcción del grafo se ejecute en tiempo de compilación, reduciendo los tiempos de ejecución del algoritmo.

Simplificación de expresiones constantes
Durante el proceso de análisis del árbol de expresiones, se realizará una búsqueda de las expresiones que sean constantes y se puedan sustituir directamente por su resultado al momento de construir el grafo.

3

Estado del arte

En la actualidad existen multitud de herramientas de *CFD* profesionales. Entre ellas destacan algunos productos comerciales como *Autodesk CFD*[10], siendo *Ansys Fluent*[11] el más extendido en el mercado. Estos programas requieren una suscripción pagada para ser utilizados, y son programas propietarios, por lo que no es posible conocer con certeza qué tipo de herramientas utilizan internamente.

La alternativa a los productos comerciales son los programas de código abierto, que son mantenidos por la comunidad o por instituciones. El programa de este tipo que más destaca es *OpenFOAM*[12], que es una de las herramientas más extendidas y utilizadas en la actualidad. Sin embargo, *OpenFOAM* presenta una limitación, pues no incluye una interfaz gráfica, lo cual es una limitación para usuarios sin experiencia en la programación.

Hay productos en el mercado cuya finalidad es proporcionar una interfaz gráfica que facilite el uso de *OpenFOAM*. Uno de estos productos es *Helyx*, desarrollado por la empresa *Engys*[13]. Este producto, además de la interfaz gráfica, incluye ciertas modificaciones y expansiones a OpenFoam, como modelos avanzados, soporte técnico, y un desarrollo activo para la inclusión de *Expression Templates*. *Helyx* es un producto comercial bajo suscripción, pero proporciona a los usuarios el código fuente. Por esta razón, este proyecto se va a realizar utilizando como base el código fuente del programa *Helyx*.

Los software de *CFD* representan los datos mediante el uso de campos (*fields*), basados en estructuras de datos lineales, implementadas como listas. Para operar con los campos se utiliza la sobrecarga de operadores[14], una herramienta de C++ que permite definir operaciones de una clase asociadas a operadores (+, -,

etc.).
La forma más común de implementar operaciones sobre campos consiste en definir las operaciones mediante operadores, y apoyarse en el uso campos temporales cuando sea necesario.
Esta técnica se basa ampliamente en el uso de herramientas que evitan copias innecesarias de memoria, como *std::unique_ptr* y *std::shared_ptr* del estándar (a partir del estándar *C++11*), o, en el caso de *OpenFOAM*, *tmp*, una implementación manual de este tipo de punteros, desarrollada previa a la existencia de las herramientas del estándar.

Existen programas como *Ansys Fluent*[15] que proporcionan herramientas basadas en arquitecturas gráficas que aprovechan el elevado grado de paralelismo nativo a las *GPU*.
Las implementaciones tradicionales de programas *CFD* incluyen muchos bucles que recorren los campos para resolver partes del problema. Para lograr aprovechar el paralelismo que ofrecen las unidades gráficas es necesario convertir cada bucle a un *kernel*, una función específica que implementa el bucle de forma paralela en la *GPU*.

Expression Templates supone una gran ventaja para estas implementaciones, ya que simplifica mucho la conversión, centralizando todos los bucles a un *kernel* único, lo que mejora mucho la mantenibilidad del proyecto. Sin embargo, para este proyecto no se van a tener en cuenta las implementaciones para *GPU*, pues el funcionamiento de *Expression Templates* es el mismo que en una implementación de *CPU*.

Expression Templates es una técnica que se presentó inicialmente en la década de 1990[16] y que permite simplificar la implementación de operaciones sobre vectores y matrices. Consiste en modificar el retorno de los operadores para que devuelvan una estructura que representa la operación en vez de devolver el resultado.
Para obtener el resultado, basta con implementar un único bucle que pide a la estructura el resultado para cada elemento, que se calcula recursivamente.
La ventaja que ofrecen frente al uso de temporales es que elimina la necesidad de crear campos temporales. Sin embargo, implementar expresiones para ciertos tipos de operaciones puede resultar complicado o imposible. Además, el código necesario para implementar ciertas operaciones con *Expression Templates* supone añadir mucha complejidad, lo que reduce la mantenibilidad del proyecto.

Otra técnica ampliamente utilizada en programas de *CFD* es el uso de paralelización a nivel de proceso, que permite dividir el problema en partes que cada proceso puede resolver de manera independiente, utilizando comunicación entre procesos donde sea necesario. Esto permite la ejecución en clústeres, reduciendo enormemente los tiempos de ejecución globales.
Para implementar la paralelización se recurre a librerías de paso de mensajes como *OpenMPI*[17], que proporcionan diversas herramientas para permitir una

comunicación eficiente entre máquinas. Si bien estas librerías permiten reducir los tiempos de ejecución de la simulación completa, la parte asignada a cada proceso se resuelve de la misma manera. Por ello, no se va a tener en cuenta el uso de esta librería para este proyecto, ya que, sin importar si se utiliza o no, la posible mejora en rendimiento será la misma.

4

Tecnologías, Herramientas y Metodologías

4.1. Tecnologías y Herramientas

Puesto que el trabajo se basa en el producto Helyx, ciertas tecnologías que se han usado vienen asociadas a este producto, pues es un programa en producción, con unos requisitos ya establecidos.

Lenguaje de programación
El lenguaje de programación utilizado es C++, más concretamente, el estándar C++17. No es el estándar más reciente, pero ofrece gran soporte para metaprogramación y facilita el desarrollo de programas utilizando esta técnica.
Por otro lado, el proyecto se apoya en la librería *OpenMPI*[17], una librería para implementar programas paralelos mediante paso de mensajes. Utiliza la versión 1.4.2.

Compilador
El compilador que utiliza *Helyx* es *emake*, un sistema de compilación y gestión de dependencias basado en `make`[18].

La versión de gcc que utiliza *make* es la que tenga instalada el sistema, en este caso `gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0`, utilizando los flags de compilación `-O2`, para añadir optimización, y `-march` para especificar la aquitectura del ejecutable
Adicionalmente, se utiliza *ninja*[19] como sistema intermediario para permitir

que la compilación se realice de forma paralela utilizando el máximo de núcleos disponibles, acelerando así el proceso de compilación.

Sistema

Debido al tamaño del proyecto, el desarrollo se ha realizado en un servidor linux dedicado, capaz de compilar el proyecto en mucho menor tiempo que un ordenador convencional. Debido a esto, se ha trabajado en remoto, mediante el uso de sesiones *SSH*.
Para ello, se ha hecho uso del programa *MobaXterm*, con la licencia de uso personal. Este programa es un cliente de *SSH* para sistemas *Windows* que, entre otras funciones, permite inicio de sesión automático, y visualización del árbol de ficheros.
Adicionalmente, permite enviar y descargar ficheros de una manera muy sencilla, e incluye un editor de texto muy ligero con coloreado de palabras clave.

Entorno de desarrollo

Puesto que el desarrollo del código se ha realizado en remoto, el programa de compilación no es uno tradicional, y la ejecución del programa tiene que realizarse de una manera específica, se ha optado por utilizar de una manera muy reducida *IDEs* (Entornos de Desarrolo Integrados) más pesados como *CLion*.
Para escribir el código se han utilizado entornos más sencillos como:
Visual Studio Code, que ofrece herramientas de análisis estático del código, junto con coloreado y tabulación automáticos, siendo muy ligero pero muy completo.
Moba Text Editor, un editor integrado en el cliente *SSH* que guarda automáticamente los cambios en remoto, y ofrece tabulación y coloreado, pero no proporciona análisis estático del código.

Por otro lado, también se han utilizado editores integrados en la terminal como *Nano* y *vim*, que se han utilizado principalmente para cambios pequeños, puesto que son editores muy rápidos y ligeros, pues no necesitan salir de la terminal.

Medición de Tiempos

Para almacenar y comparar los tiempos medidos se ha utilizado una hoja de cálculos Excel, ya que permite almacenar los tiempos de manera sencilla al mismo tiempo que tiene métodos para hacer análisis de los datos y generar gráficos sobre ellos.

Control de versiones

Helyx utiliza *git* como herramienta de control de versiones. Para este proyecto se ha utilizado una rama local de dicho proyecto.

4.2. Metodología

Este proyecto se ha dividido en varias fases para las que se ha optado por utilizar una metodología diferente, adecuada al objetivo de cada fase.

Estudio de *OpenFoam* y *Expression Templates*
Análisis de *Helyx*, *OpenFoam* y el entorno de *CFD* en general, su implementación y estructura del proyecto. Investigación sobre *Expression Templates* y su relación con *OpenFoam*. Estudio del patrón *CRTP* y cómo se aplica a *Expression Templates*. Desarrollo de pruebas de concepto de manipulación de *Expression Templates*.

Investigación del algoritmo
Posibles algoritmos de optimización de código y expresiones. Investigación del algoritmo del Grafo Dirigido Acíclico, análisis de las posibles implementaciones. Estudio de ventajas e inconveniences de integrar el algoritmo en el entorno de *Expression Templates*.

Implementación del algoritmo
Análisis de la implementación para tiempo de compilación y sus ventajas y contratiempos. Implementación del algoritmo de sustitución de expresiones simples. Desarrollo del algoritmo utilizando una estructura de clases basada en polimorfismo.

Análisis de rendimiento
Desarrollo de un programa de pruebas para medir el rendimiento del algoritmo. Análisis del rendimiento del Grafo Dirigido Acíclico frente a *Expression Templates*.

Revisión del algoritmo
Revisión del algoritmo tras el análisis de rendimiento. Reprogramación en caso necesario para evitar sobrecarga del *Dynamic Dispatching*.

Análisis de rendimiento y conclusiones finales
Estudio del rendimiento tras la revisión del algoritmo. Comparativa a fondo en un caso de uso realista. Análisis del cumplimiento de los objetivos originales y conclusiones.

5

Descripción Informática

5.1. Requisitos

Para describir formalmente los objetivos que debe cumplir la librería, se han desarrollado los siguientes requisitos, tanto funcionales como no funcionales.

Requisito 1 - La librería no deberá afectar al uso de *Expression Templates*
La librería deberá ser totalmente transparente para un usuario que quiera utilizar *Expression Templates*. No se deberá modificar el formato de los datos de entrada ni los de salida, únicamente cómo se procesan para que se realice de una manera más eficiente.

Requisito 2 - La librería deberá mejorar el rendimiento de *Expression Templates*
Utilizar la librería deberá ofrecer una mejora de rendimiento frente a no usarla para justificar su implementación de un entorno de producción,.

Requisito 3 - La librería deberá ser modular e independiente
La librería deberá implementarse como un módulo independiente, que pueda ser añadido o retirado del paquete de *Helyx* de una forma simple, minimizando su impacto en la complejidad del proyecto.

Requisito 4 - La librería deberá soportar todos los tipos de expresiones
La librería deberá proporcionar una implementación que se adapte a todos los

tipos de expresiones que existen en el proyecto.

Requisito 5 - La librería deberá poder adaptar su funcionamiento para expresiones concretas
La librería deberá ofrecer un sistema que permita a un programador adaptar el funcionamiento para que optimice su funcionamiento en expresiones concretas que tengan un comportamiento especial.

5.2. Arquitectura y Análisis

El código está diseñado como una librería, de forma que se pueda añadir al programa principal sin mayores complicaciones. Los cambios que se deben hacer al código principal, se explicarán en la sección de Diseño e Implementación.

La librería incluye los siguientes ficheros:
El fichero principal es `adg.H`, que incluye las clases y funciones principales.
`tupleTools.H` es un fichero auxiliar que incluye funciones para facilitar el trabajo con tuplas.
El fichero `gradient.H` incluye clases y funciones para implementar la expresión correspondiente a la operación gradiente. Esta expresión se ha diseñado como un ejemplo de un caso de uso de ExpressionTemplates realista, con una gran carga computacional.
El fichero `is_1D.H` contiene las estructuras que permiten identificar expresiones constantes

El fichero `expressionTemplates.H`, si bien parte del proyecto original, es el fichero en cuyo contenido se basa la librería, y sobre el que se ha realizado algún cambio necesario.
El fichero principal de la librería es `gda.H`, donde aparecen las definiciones de las clases y funciones de la librería. No se han implementado ficheros con extensión `.C` debido a que todas las clases y funciones hacen uso intensivo de plantillas, cuyo uso se dificulta cuando se separa el código en declaración y definición.

5.3. Diseño e Implementación

5.3.1. Sustitución de expresiones constantes

Anteriormente se ha mencionado que *Expression Templates* tan solo convierte la expresión que escribe un programador en una estructura sobre la que se puede operar.
Previo a la construcción de la expresión, el compilador es capaz de detectar par-

tes de la expresión que son obvias (a + x - x), y las elimina. Sin embargo, las optimizaciones que el compilador es capaz de realizar se limitan a los símbolos que aparecen en la expresión, sin tener en cuenta lo que significan.
El compilador no es capaz de simplificar operaciones entre campos 1D, que cumplen con la condición de que representan un valor único, o constante a lo largo de todo el campo. Las operaciones entre campos 1D se pueden sustituir por su resultado en el momento de la creación de la expresión, pues los campos almacenen un único valor.

Realizar esta optimización parte de poder identificar qué expresiones cumplen con la condición de ser 1D. Esto se podría realizar añadiendo un booleano estático que represente la condición en el propio código de cada expresión, sin embargo, puesto que uno de los requisitos del trabajo es minimizar los cambios al código base, se ha descartado esta opción.
Por ello, se ha diseñado un *struct* que representa la condición de validez de cualquier expresión.

```
1 template<typename T>
2 struct is_1D{
3     constexpr static bool value = false;
4     static void run(){printf("Default");}
5 };
```

Código 5.1: struct *is_ 1D*

El *struct* se define como una plantilla, en la que el caso general define una variable booleana inicializada a falso.
Para las expresiones hoja que sí son 1D, se define una especialización de la plantilla, en la que la variable booleana se inicializa a verdadero.

Puesto que las expresiones recursivas se implementan utilizando el patrón *CRTP*, la especialización de la expresión general no funciona correctamente, pues las expresiones utilizan el tipo que hereda de la expresión general. Implementar una especialización para cada subclase no es una opción viable, pues las subclases se generan mediante una macro integrada en el núcleo del programa, por lo que hacer una implementación para cada subclase incumple uno de los requisitos del proyecto.

Otra forma de lograr conocer si una expresión recursiva es simple, es extrayendo la superclase a la que pertenece la expresión. A pesar de que este método también va a requerir hacer un cambio en el núcleo de la aplicación, el cambio es mucho más superficial, por lo que, a pesar de incumplir el requisito, es la opción más viable:

```
1 using motherclas = binaryExpression<valueT, LHS, RHS, subT>
```

Código 5.2: Declaración *motherclass* en *binaryExpression*

Este fragmento de código se añade a todas las expresiones, adaptándola, de forma que en las expresiones hoja devuelva la propia clase, y en las expresiones recursivas devuelva la superclase (binaria, unaria, etc.).

Sustitución de las expresiones
Sustituir las expresiones 1D por su resultado requiere modificar las expresiones recursivas, para que, en lugar de almacenar la expresión, computen directamente el resultado al momento de crearlas, almacenando únicamente el valor resultante de la operación.
Esto requiere, inevitablemente, modificar el código asociado a las expresiones recursivas para que, en caso de que sus partes izquierda y derecha sean 1D, se cree de forma modificada.

Para implementar esto, se ha hecho uso de *SFINAE* (*Substitution Failure Is Not An Error*, fallo en la sustitución no es un error), una técnica mediante la cual se pueden implementar plantillas que se seleccionan o descartan según una condición, que debe conocerse en tiempo de compilación. Utilizando esta técnica, se crea una nueva versión de las expresiones recursivas simplificables que modifica el constructor para que compute el resultado y lo almacene, y el *operator[]* para que devuelva el valor almacenado.

```
1 template<typename value_T, typename LHS, typename RHS, typename SubT
     >
2 struct binaryExpression<value_T, LHS, RHS, SubT, std::enable_if_t<
     is_1D<typename LHS::motherclass>::value && is_1D<typename RHS::
     motherclass>::value>> :
3 fieldExpressionImpl<binaryExpression<value_T, LHS, RHS, SubT> >
```

Código 5.3: Declaración de *binaryExpression* con *is_ 1D*

Una vez realizados estos cambios de una manera similar para las expresiones unarias, y realizadas las pruebas que aseguran que este código funciona como es esperado, se puede analizar la mejora de rendimiento frente a la técnica de *Expression Templates* por sí sola. Este análisis de rendimiento se realizará en el capítulo 6, donde se valorará su viabilidad de cara al futuro.

5.3.2. Algoritmo

El algoritmo que se ha empleado para la optimización de expresiones es una adaptación del algoritmo del Grafo Dirigido Acíclico que aparece en el libro *Compiladores: Principios, técnicas y herramientas*[20].

El algoritmo consiste en la transformación de un árbol con subárboles repetidos a un grafo dirigido acíclico en el que cada nodo aparezca una única vez. Para lograr este objetivo, se debe recorrer el árbol mediante un recorrido en postorden, y por cada nodo, añadirlo al grafo bajo una condición:

- Si el nodo no ha aparecido previamente en el recorrido, se añade al grafo un nuevo nodo que represente al nodo del árbol.
- Si ya ha aparecido, se ignora.

Una vez se añade el nodo al grafo, se buscan sus hijos en el grafo, y se añaden las referencias correspondientes.
El resultado que se obtiene una vez se completa el algoritmo es una estructura en la que un nodo puede tener más de un padre (dos nodos pueden compartir un mismo nodo hijo), habiendo convertido el árbol en un grafo. Adicionalmente, puesto que cada nodo solo necesita conocer a sus hijos y no a sus padres, la estructura resultante es un grafo dirigido acíclico.

Si se pretende optimizar la siguiente expresión utilizando el algoritmo:

$$a * (b - c) + (b - c) * d$$

El árbol resultante sería el siguiente: Visualmente, se puede observar que el subár-

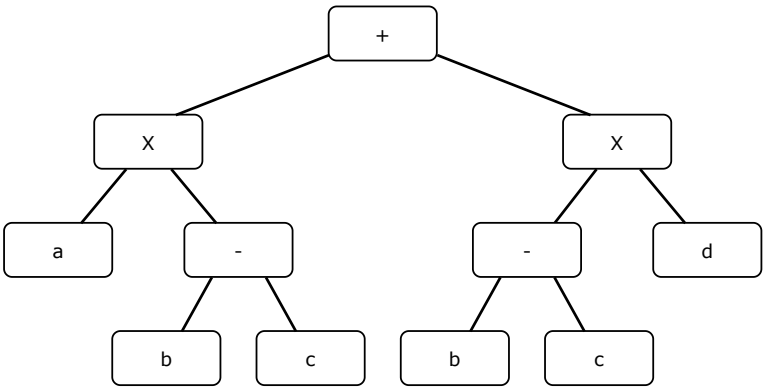

Figura 5.1: Árbol correspondiente a la expresión superior.

bol (b - c) aparece dos veces, y, por tanto, es una operación que se puede optimizar. Por ello, el grafo resultante una vez se aplique el algoritmo, debe ser similar al árbol, pero eliminando uno de los subárboles correspondientes a la resta (b-c), y referenciando los 2 nodos que lo contienen hacia el mismo nodo.
El grafo que se obtiene una vez ejecutado el algoritmo sobre el grafo anterior es el siguiente:

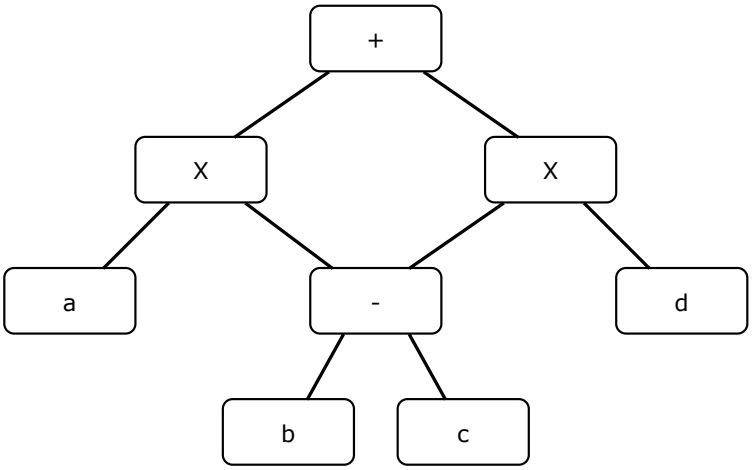

Figura 5.2: GDA Correspondiente a la expresión superior.

El libro sugiere que para implementar este algoritmo se utilice una tabla hash con encadenamiento separado, pues es una estructura de datos en la que la complejidad temporal de las búsquedas es constante.
Sin embargo, dada la naturaleza de las expresiones para las que se va a aplicar el algoritmo, se ha optado por utilizar una lista (std::vector de C++), ya que las expresiones tienen tamaños reducidos, y una búsqueda lineal en una lista pequeña es tan eficiente como su implementación en una tabla hash en el mismo contexto.

5.3.3. Construcción del Grafo en tiempo de compilación

Puesto que *Expression Templates* representa las expresiones mediante una estructura que se construye en tiempo de compilación, es viable plantear como hipótesis realizar el proceso de construcción del grafo en tiempo de compilación. Si bien la construcción del grafo no es una operación que consuma mucho tiempo, trasladar este cómputo a tiempo de compilación podría potenciar el rendimiento del algoritmo general, especialmente en situaciones que requirieran construir un mismo grafo más de una vez.
Para probar o refutar esta hipótesis, se puede plantear un pseudocódigo, o una lista de pasos que se deberían seguir para completar esta implementación, y buscar una contradicción o una imposibilidad. De esta forma, si se llega a una imposibilidad quedaría refutada la hipótesis.

Para implementar la construcción del grafo sería necesario hacer uso de la palabra reservada `constexpr`, un modificador que se puede aplicar a declaraciones, condiciones y funciones, y que indica al compilador que el segmento que modifica se debe evaluar en tiempo de compilación. Adicionalmente, la estructura de datos que se utilizaría es la tupla, una estructura de datos lineal y estática, que se puede utilizar completamente en tiempo de compilación.

Cuando el algoritmo procesa un nodo, comprueba si ha aparecido antes en el grafo, añadiéndolo únicamente en caso de no haber aparecido anteriormente. Esto supone una imposibilidad para las expresiones hoja de tipo *referenceExpression*, que representan una lista, y cuya igualdad se avalúa comparando punteros. Los punteros existen únicamente en tiempo de ejecución, por lo que comparar este tipo de expresiones en tiempo de compilación resulta imposible.

Otra imposibilidad surge en el procesamiento de nodos intermedios. Los elementos que contiene una tupla aparecen como parámetros de plantilla de la propia tupla, por lo que hacer una función que añada elementos condicionalmente a una tupla implica que el retorno de la función varía condicionalmente. En c++ una función puede devolver un único tipo, por lo que resulta muy complejo resolver este problema.

Debido a estas razones, se ha decidido descartar esta hipótesis, optando por construir el grafo en tiempo de ejecución, y asumiendo el coste computacional que esta operación implica.

5.3.4. Implementación basada en Polimorfismo Dinámico

Para implementar el grafo no es necesario recurrir a una clase compleja que almacene atributos y métodos, pues la funcionalidad que se necesita se reduce a la creación del grafo, y la evaluación de la expresión.
Utilizar una implementación basada en un vector es la opción más sencilla, minimizando la cantidad de código que requiere, mejorando así la mantenibilidad.
Debido a que los elementos están basados en plantillas, no se puede utilizar un vector de elementos directamente, ya que requeriría conocer el tipo concreto de todos los elementos en tiempo de compilación. Por ello, se ha implementado la clase *TEW (Type Eraser Wrapper)*, un envoltorio simple cuyo objetivo es ocultar (borrar) la genericidad de la clase que envuelve.
Utilizar este envoltorio tiene ventajas y contratiempos. La ventaja principal, como se ha mencionado anteriormente, es la habilidad de poder utilizar clases plantilla como elementos de un vector. Sin embargo, utilizar este envoltorio también supone un problema: eliminar los tipos de las plantillas implica que se pierde la capacidad de conocer el tipo exacto de cada elemento, lo cual requiere de otras técnicas que permitan acceder a estos tipos.

Para implementar los elementos del grafo, se ha optado por una clase genérica de la que se implementan especializaciones parciales para las expresiones recursivas. De esta forma, las expresiones hoja que no necesitan optimización utilizan la implementación general, y el resto de expresiones utilizan la implementación de su clase madre. Esta implementación depende del uso del polimorfismo dinámico, ya que se utiliza una clase madre (*TEW*) de la que heredan varias clases con múltiples implementaciones para sus métodos (*ADGElement<>*)

Utilizar esta implementación implica que los elementos no se pueden crear utilizando directamente el tipo de cada expresión, sino que los elementos del grafo deben crearse usando la clase madre (*binary*, *unary*). Esto es debido a que, en una clase genérica, una implementación parcial para una clase que utiliza el patrón *CRTP*, como es el caso de las *Expression Templates*, no funciona para las clases que heredan de ella.

De esta forma, el diagrama de clases utilizadas en la construcción del grafo es el siguiente:

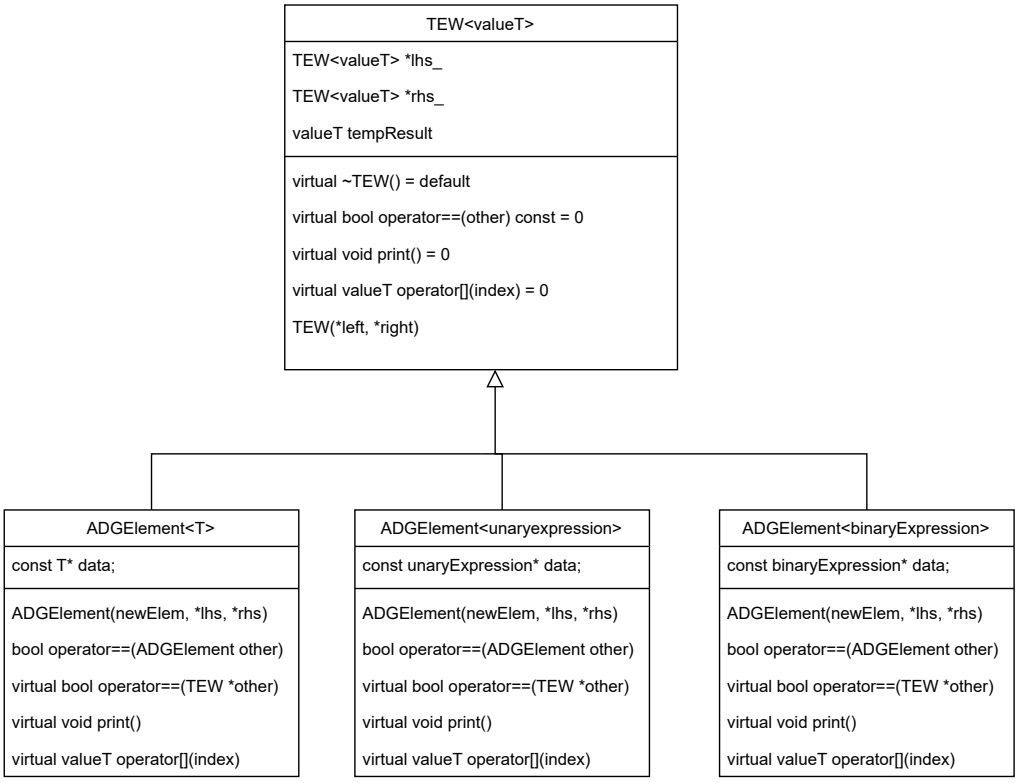

Figura 5.3: Diagrama de clases utilizadas en la construcción del grafo.

Todos los métodos de *TEW* se han declarado utilizando la palabra reservada *virtual*. Definir un método de esta manera indica al compilador que es un método polimórfico, y cuando sea referenciado se deberá comprobar la clase exacta sobre la que se llama al método para saber la implementación que se debe utilizar. De esta forma, se permite que todos los elementos del grafo compartan la cabecera para sus métodos, proporcionando implementaciones adecuadas.

La clase *ADGElement* es una clase genérica que hereda de *TEW*, y proporciona tanto una implementación general, como implementaciones para *unaryExpression* y *binaryExpression*.

Para permitir una interacción polimórfica entre todos los elementos del grafo, el método *operator==* debe poder recibir un puntero a un objeto *TEW*. Puesto que el operador de igualdad de una expresión sólo permite recibir otra expresión del mismo tipo, cada especialización de *ADGElement* debe hacer un cast dinámico para poder probar la igualdad entre objetos del mismo tipo.
En el código 5.4 se puede observar este comportamiento en la especialización para *binaryExpression*.

```
1    virtual bool operator==(TEW<valueT>* other) const override{
2        auto otherModified =
3                dynamic_cast<GDAElement<valueT, binaryExpression<
                    valueT, LHS, RHS, subT>>*>(other);
4        if(otherModified)return data->operator==(*(otherModified->
             data));
5        return false;
6    }
```

Código 5.4: *operator==* de *ADGElement* para *binaryExpression*

La implementación general de la clase *ADGElement* está diseñada para ser empleada por expresiones hoja, o expresiones que no se quieran optimizar. Por esta razón, el método *operator[]* llama al propio operador de la expresión, sin realizar optimizaciones.
Las optimizaciones en el operador corchete se deben implementar en las especializaciones parciales de esta clase, actuando adecuadamente a cada caso específico. Esto se puede observar en la especialización de *binaryExpression*, que hace uso de los resultados almacenados de los elementos izquierdo y derecho que almacena el propio elemento como miembros de la clase:

```
1    virtual valueT operator[](const label index){
2        return static_cast<const subT *>(data)->run(this->lhs_->
             tempResult, this->rhs_->tempResult);
3    }
```

Código 5.5: *operator[]* de *ADGElement* para *binaryExpression*

Construcción del Grafo

Para construir el grafo, se ha implementado una función sobrecargada que adopta una definición adecuada para cada tipo de expresión. Es suficiente incluir únicamente una definición para los tipos generales y, aunque la función se llame con una expresión con tipo derivado como parámetro, se ejecuta la definición correcta.
Las implementaciones de las expresiones hoja son triviales, únicamente crean el vector con un elemento: un *ADGElement* de la expresión.

Además de las expresiones hoja, se han incluido dos implementaciones para las expresiones unarias y binarias respectivamente:

```
1  template<typename valueT, typename LHS, typename RHS, typename subT>
2  std::vector<TEW<valueT> *> factorize(const binaryExpression<valueT,
       LHS, RHS, subT> &expr){
3      auto tmp1 = factorize(expr.lhs_);
4      auto tmp2 = factorize(expr.rhs_);
5
6      auto left = tmp1.back();
7      auto right = tmp2.back();
8
9      // Join tmp2 into tmp1
10     for(TEW<valueT> * i : tmp2){
11         bool found = false;
12         for(TEW<valueT> * j : tmp1)
13             if(i->operator==(j)){
14                 found = true;
15                 if(i->operator==(right))right = j;
16             }
17         if(!found){
18             TEW<valueT> *elemLeft = i->lhs_;
19             TEW<valueT> *elemRight = i->rhs_;
20             for(TEW<valueT>* j : tmp1){
21                 if(j->operator==(elemLeft))elemLeft = j;
22                 if(j->operator==(elemRight))elemRight= j;
23             }
24             i->lhs_ = elemLeft;
25             i->rhs_ = elemRight;
26             tmp1.push_back(i);
27         }
28     }
29     tmp1.push_back(new GDAElement<valueT, binaryExpression<valueT,
           LHS, RHS, subT>>(&expr, left, right));
30     return tmp1;
31 }
```

Código 5.6: Construcción del grafo para *binaryExpression*

Si bien el código 5.6 muestra la factorización de expresiones binarias, la función para expresiones unarias funciona de una manera análoga.
Esta implementación se basa en generar dos grafos para las partes izquierda y derecha de la expresión y unirlos en uno general, ignorando los elementos que se repitan entre ellos. Por último, añade un nodo para la propia expresión, raíz del árbol, y devuelve el vector.

Para realizar el recorrido postorden del árbol de la expresión, se factoriza primero el hijo izquierdo, después se une el hijo derecho al izquierdo, y por último se añade el elemento asociado a la propia expresión.
Las variables *left* y *right* representan los hijos directos del nodo que se está factorizando. Se inicializan a los últimos elementos de cada vector, pues esas son las raíces de los subárboles izquierdo y derecho. En caso de que el subárbol derecho también aparezca como parte del izquierdo, se cambia el elemento al que apunta este puntero al correspondiente del grafo izquierdo. Cabe destacar la línea 29,

donde se crea el elemento de la expresión de tipo expresión binaria, no del tipo específico de la expresión que se está factorizando.

Evaluación

Para facilitar la evaluación de las expresiones mediante el grafo dirigido acíclico, se ha creado la clase *ADGList*, que facilita realizar las operaciones y permite proporcionar la misma interfaz que *Expression Templates*.
Puesto que el grafo se construye mediante un recorrido postorden de la expresión, los elementos aparecen en el vector de forma que un elemento nunca aparece antes que un nodo perteneciente a un nivel superior del árbol. Esto permite evaluar el resultado de la expresión mediante un recorrido del vector, evaluando el resultado intermedio de cada elemento.
Una vez se ha recorrido el vector, se devuelve el resultado temporal del último elemento del vector. Puesto que este nodo representa la raíz del árbol, su resultado intermedio es igual al resultado de evaluar la expresión.

La implementación de *Expression Templates* proporciona la función *assignFieldExpression* como interfaz para evaluar expresiones. Para proporcionar una interfaz similar, se ha desarrollado la función *assignFieldTEW*, que recibe los mismos argumentos que su análoga.
Esta función es muy similar a *assignFieldExpression*, con la diferencia de construir el grafo previo al bucle que evalúa la expresión.

Debido a que la interacción que hay entre los elementos del grafo se realiza únicamente mediante el envoltorio *TEW*, todas las llamadas que se producen tanto en la construcción del grafo, como en la evaluación de las expresiones dependen del enlace dinámico (*Dynamic Dispatching*), mecanismo mediante el que se resuelven los tipos polimórficos en tiempo de ejecución. Depender del *Dynamic Dispatching* implica que en todas las llamadas que se hacen a métodos de las variables de tipo *TEW* hay una sobrecarga implícita que añade tiempo extra a cada llamada.

Esto puede significar tiempos de ejecución más largos, llegando a comprometer la eficacia del uso de este método en casos generales, por ello será de suma importancia hacer pruebas extensivas que demuestren si el balance de beneficios y contratiempos de esta implementación merece la pena.

Para evitar el sobrecoste que implica el uso del polimorfismo dinámico, se ha planteado una implementación que evita el uso de este recurso, recurriendo a técnicas más complejas en su lugar, que aumentan la complejidad del código, pero reducen el tiempo de ejecución.

En el capítulo 6 se analizarán ambas implementaciones, y se valorará la opción más viable a largo plazo.

5.3.5. Implementación basada en polimorfismo estático

Una vez completada la implementación utilizando *Dynamic Dispatching* y realizado su análisis de rendimiento, se ha optado por hacer otra implementación del algoritmo utilizando un enfoque distinto. Esta segunda implementación tiene el objetivo de eliminar el sobrecoste que supone depender del *Dynamic Dispatching*.

Por ello, el objetivo principal de la segunda implementación es diseñar el grafo de manera que se sustituyan las instrucciones que dependen del *Dynamic Dispatching* por otras que no tengan esta limitación.

La ventaja principal que se busca con lograr esta implementación es reducir el tiempo que se tarda en evaluar el grafo, mejorando así el rendimiento global. Sin embargo, esta implementación también supone un coste, no para la ejecución del programa, sino para el programador que lo desarrolla y mantiene. La ventaja que ofrecía delegar la deducción de los tipos al lenguaje mediante *Dynamic Dispatching* era la relativa simplicidad en el código. Realizando una especialización de los elementos del grafo para las expresiones generales (*binary*, *unary*, etc.), se lograba que la implementación funcionara para todas las expresiones.

Cuando se elimina el uso del *Dynamic Dispatching*, la complejidad que supone deducir el tipo de cada expresión adecuadamente queda delegada en el programador. Esto se ve reflejado como una mayor complejidad en el código, resultando en peor mantenibilidad y depuración.

La manera de evitar el uso de este recurso es permitir una especialización de los elementos del grafo para cada tipo de expresión. Así, el algoritmo de construcción del grafo puede crear los elementos utilizando directamente los tipos de cada expresión, eliminando la necesidad de utilizar *dynamic_ cast*, pudiéndose sustituir por *static_ cast*.

Una manera de implementarlo sería programar manualmente cada especialización de los elementos, añadiéndola en las macros de generación de expresiones. Sin embargo, realizar la implementación de esta manera incumpliría uno de los requisitos del proyecto, que dictamina que se minimizarán las modificaciones al código base, por lo que se descarta esta implementación.

Para lograr este objetivo sin incumplir el requisito se puede hacer uso *SFINAE (Substitution Failure Is Not An Error)*, una técnica avanzada que permite implementar especializaciones parciales de plantillas según una condición que se conozca en tiempo de compilación.
Utilizando *SFINAE*, se puede crear una especialización parcial la clase *ADGElement* que aplique a todas las clases que hereden de una cierta expresión general como *unary* o *binary*.

El diagrama de clases para esta implementación es el siguiente:

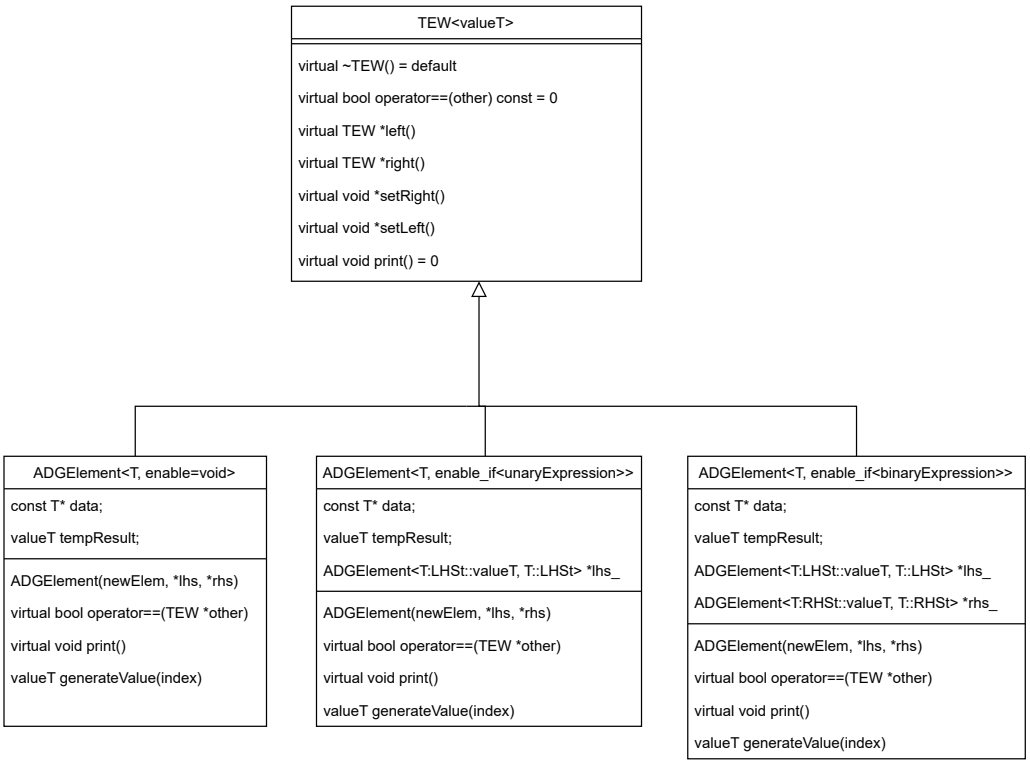

Figura 5.4: Diagrama de clases haciendo uso de *SFINAE*

En esta implementación se observan varias diferencias principales, entre las más destacables se encuentran: la variable que almacena el resultado temporal no se encuentra en el envoltorio, sino en el propio elemento; los punteros a los elementos izquierdo y derecho también se han reubicado, perteneciendo ahora a la clase *ADGElement*, y apareciendo únicamente en las especializaciones de expresiones recursivas; se han añadido métodos virtuales que facilitan la construcción del grafo.

Evaluación

Puesto que esta implementación modifica la estructura general del grafo, la evaluación de las expresiones no se puede realizar de la misma manera.
La implementación que depende de *Dynamic Dispatching* evalúa las expresiones recorriendo el vector que representa el grafo, ejecutando en cada elemento el método que genera su resultado temporal.
Esta forma de evaluar el grafo provoca que todas las llamadas al *operator[]* se produzcan sobre un objeto de tipo *TEW*, por lo que este método debe ser virtual.

Para poder ejecutar dicho método sin utilizar *Dynamic Dispatching*, sería necesario conocer el tipo concreto de cada elemento del grafo para poder hacer

un *static_cast*.
Por ello, para evaluar la expresión en esta implementación, se ha optado por utilizar una estrategia recursiva. En esta estrategia, es cada elemento quien llama al método de sus hijos, imitando la manera que utiliza *Expression Templates* para evaluar el árbol.

La ventaja de esta estrategia es que únicamente se necesita el último elemento del vector, que representa la raíz del árbol, eliminando la necesidad de tener una clase que represente el grafo. Sin embargo, este método también conlleva una desventaja, puesto que con esta implementación, siempre que se ejecuta el método de generar valor, se debe comprobar si el resultado se ha calculado previamente o no.

En expresiones complejas, donde computar el resultado es más costoso que hacer la comprobación, sigue mereciendo la pena cachear el resultado. Sin embargo, en expresiones muy simples (sumas, restas, etc.), donde calcular el resultado es más rápido que comprobar si el resultado se ha computado anteriormente, no merece la pena cachear los resultados intermedios.

Se ha añadido una condición que permite decidir en tiempo de compilación si se quiere cachear el resultado intermedio de una cierta especialización del elemento.

ADGElement

Utilizando *SFINAE*, se ha implementado especializaciones parciales de *ADGElement* que admiten todas las expresiones que heredan de *unaryExpression* y *binaryExpression* respectivamente. El código para expresiones binarias es el siguiente:

```
1
2  template<typename valueT, typename T>
3  struct ADGElement<valueT, T,
4          typename std::enable_if_t<
5                  std::is_base_of_v<unaryExpression<valueT, typename T
                       ::LHSt, T>, T> &&
6                  !differentADGBehaviour<T>::value>> : public TEW{
7
8      const T *data;
9
10
11     ADGElement<typename T::LHSt::valueT, typename T::LHSt> *lhs_;
12
13     label lastIndex;
14     valueT tempValue;
15
16     ADGElement(const T *newElem, TEW *lhs, TEW *rhs)
```

```
17          : data(newElem), lhs_(static_cast<ADGElement<typename T::
                LHSt::valueT, typename T::LHSt> *>(lhs)), lastIndex(-1)
                {}
18
19      virtual bool operator==(TEW *other) const final override{
20          auto otherModified =
21                  dynamic_cast<ADGElement<valueT, T>*>(other);
22          if(otherModified)return data->operator==(*(otherModified->
                data));
23          return false;
24      }
25
26      valueT generateValue(const label index) {
27  #ifdef CACHING_ADG
28          if(lastIndex != index){
29              lastIndex = index;
30              tempValue = data->run(lhs_->generateValue(index));
31          }
32          return tempValue;
33  #else
34          return data->run(lhs_->generateValue(index));
35  #endif
36      }
37
38      virtual TEW *left() final { return lhs_;}
39
40      virtual void setLeft(TEW *left) final {lhs_ = static_cast<
            ADGElement<typename T::LHSt::valueT, typename T::LHSt> *>(
            left);}
41
42      virtual void print() final {
43          printTuple(std::make_tuple(*data));
44      }
45  };
```

Código 5.7: Especialización de *ADGElement* para *binaryExpression*

Para poder aplicar *SFINAE*, se ha empleado los rasgos de tipo (*type traits*), *structs* que permiten conocer información sobre tipos en tiempo de compilación. Concretamente, se ha empleado `std::enable_if`, un *structs* que define un tipo solo si se cumple una condición (la condición se debe resolver en tiempo de compilación).
Utilizando `std::enable_if`, se define una plantilla que solo es válida si se cumple una condición. De esta forma, si se cumple la condición, la plantilla es completamente válida y el compilador la tiene en cuenta a la hora de asignarla. Por otro lado, en caso de no cumplirse dicha condición, `std::enable_if` no define el tipo, y se produce un fallo cuando el compilador intenta crear una instancia. Debido a *SFINAE*, lo que se produce es un fallo y no un error, por lo que el compilador, en vez de terminar la compilación, únicamente descarta la plantilla.

Utilizando esta técnica en conjunción con *std::is_ base_ of*, que indica si una

clase hereda de otra, se define una especialización de la clase que únicamente es válida cuando el tipo que recibe es una clase derivada de *binaryExpression*.

Cuando se quiera implementar *ADGElement* para otros tipos de expresión abstractos, se podrá crear una especialización parcial de manera similar a como se ha definido para expresiones binarias.
Esta implementación tiene capacidad de permitir especializaciones completas además de las parciales. Para realizarlas, únicamente se crea la especialización ignorando el tipo dedicado a *SFINAE*, que en la plantilla general aparece definido con un valor por defecto.

Construcción del grafo

Debido al cambio realizado en los elementos del grafo, se deben hacer modificaciones a las funciones de factorización de expresiones. La cabecera de la función se mantiene para esta implementación, pues sigue funcionando con las sobrecargas para expresiones generales. La diferencia principal que aparece en estas funciones es la forma de manejar los elementos, cuyos métodos han cambiado.

```
1  template<typename valueT, typename LHS, typename RHS, typename SubT>
2  std::vector<TEW *> factorize(const binaryExpression<valueT, LHS, RHS
       , SubT> &expr){
3      auto tmp1 = factorize(expr.lhs_);
4      auto tmp2 = factorize(expr.rhs_);
5
6      auto left = tmp1.back();
7      auto right = tmp2.back();
8
9      for(TEW *i : tmp2){
10         bool found = false;
11         for(TEW *j : tmp1)
12             if(i->operator==(j)){
13                 found = true;
14                 if(i->operator==(right))right = j;
15                 break;
16             }
17         if(!found){
18             TEW *elemLeft = i->left();
19             TEW *elemRight = i->right();
20             bool foundLeft = elemLeft == nullptr, foundRight =
                   elemRight == nullptr;
21             for(TEW *j : tmp1){
22                 if(j->operator==(elemLeft)){
23                     elemLeft = j;
24                     foundLeft = true;
25                 }
26                 if(j->operator==(elemRight)){
27                     elemRight= j;
28                     foundLeft = true;
29                 }
```

```
30                  if(foundLeft && foundRight)break;
31              }
32              i->setLeft(elemLeft);
33              i->setRight(elemRight);
34              tmp1.push_back(i);
35          }
36      }
37      auto *casted = static_cast<const SubT *>(&expr);
38      tmp1.push_back(new ADGElement<valueT, SubT>(casted, left, right)
            );
39      return tmp1;
40  }
```

Código 5.8: Construcción del grafo para *binaryExpression* con polimorfismo estático

Se puede apreciar uno de los cambios en las líneas 31 y 32, donde se realiza una llamada a un método en lugar hacer de la asignación. Esto garantiza una asignación correcta de los tipos cuando los objetos se desenvuelven desde el *TEW*.

Evaluación

En esta implementación no se puede recorrer el vector generando los resultados parciales de cada elemento. Hacerlo implica que en cada llamada se depende del *Dynamic Dispatching*, por lo que en esta implementación se ha cambiado el flujo de ejecución para poder evitarlo.
Debido al cambio en la manera de calcular el resultado, ya no tiene sentido utilizar la clase *ADGList*, por lo que se ha eliminado por completo.
Únicamente se ha mantenido la función de asignación, que se ha cambiado de nombre para que sea más adecuado a esta implementación.

```
1  template<class Type, typename exprT>
2  void assignFieldFactorized(UList<Type> & lst, const wrappedET<Type,
       exprT>& expr){
3      auto list = factorize(expr);
4      ADGElement<Type, exprT> *elem = static_cast<ADGElement<Type,
           exprT> *>(list.back());
5      for(label i = 0; i < expr.size(); ++i){
6          lst[i] = elem->generateValue(i);
7      }
8  };
9
10 template<class Type, typename exprT>
11 void assignFieldFactorized(UList<Type> & lst, const wrappedET<Type,
       exprT>& expr, TEW *tew){
12     ADGElement<Type, exprT> *elem = static_cast<ADGElement<Type,
           exprT> *>(tew);
13     for(label i = 0; i < expr.size(); ++i){
14         lst[i] = elem->generateValue(i);
15     }
```

```
16 };
```

Código 5.9: Función *assignFieldFactorized*

En este caso, únicamente se necesita llamar al método del último elemento del vector, que representa la raíz del árbol, que además, devuelve el valor en lugar de almacenarlo en un atributo.
Adicionalmente, se ha añadido una versión sobrecargada que recibe un *TEW*. Esta versión permite que, si una expresión se declara una vez, pueda evaluarse múltiples veces generando el grafo una única vez.
Las cualidades del problema hacen que no siempre sea posible hacer uso de esta generación previa, pero en caso de ser posible, proporcionará un rendimiento ligeramente mayor que una generación repetitiva del grafo.

En las implementaciones no aparecen los destructores para facilitar la lectura del código. Estos se definen liberando la memoria de los elementos creados mediante *new*.

6

Análisis de rendimiento

Una vez implementado el algoritmo de optimización de expresiones, se procede a analizar su rendimiento.
Para ello, se han diseñado casos de prueba que permitan realizar una comparación entre el tiempo que tarda en calcular el resultado el método de *Expression Templates*, y el Grafo Dirigido Acíclico.
Los programas de prueba se han diseñado para que funcionen utilizando un único hilo de ejecución, dado que el método de paralelización de *Helyx* se fundamenta en repartir un problema de mayor tamaño en nodos que realizan operaciones sobre *Expression Templates*. Esto significa que cada nodo podría realizar las operaciones de manera secuencial, y añadir concurrencia a este sistema no afecta al trabajo sobre *Expression Templates*.

6.1. Hardware empleado

Todas las pruebas se han realizado en el mismo servidor, que se encontraba ocioso en todos los casos. Dicho servidor cuenta con las siguientes características:

- Procesador: AMD ThreadRippper 2990WX
- Núcleos del procesador: 32
- Hilos por núcleo: 2
- Velocidad Base del procesador: 2200 MHz
- Velocidad Máxima del procesador: 3000 MHz
- Cache L1-datos: 1 MiB (32 instancias)

- Cache L1-instrucciones: 2 MiB (32 instancias)
- Cache L2: 16 MiB (32 instancias)
- Cache L3: 64 MiB (8 instancias)
- Memoria RAM: 128 GiB (125 utilizables)

6.2. Sistema de medición de tiempos

Para medir el tiempo que tarda en ejecutarse un bloque de código en C++ existen varias opciones, entre las que se encuentran las funciones *time* y *clock* de la librería *ctime*, o las herramientas de la librería *chrono*.
Para este trabajo se ha optado por utilizar la función *clock*, disponible en la librería *ctime*. Esta decisión se debe a varios factores: la función *time*, si bien es fiable útil en programas multi hilo, se ha descartado debido a su escasa precisión, pues solo mide el tiempo en segundos.
Por otro lado, se ha elegido *clock* frente a *chrono* debido a que, a pesar de que las herramientas de *chrono* ofrecen gran precisión, *clock* no mide tiempo real, sino el tiempo de CPU, más concretamente, el número de ciclos del procesador que el programa ha ejecutado.
Puesto que el objetivo es analizar el rendimiento de *Expression Templates* frente al Grafo Dirigido Acíclico, medir el uso de *CPU* del programa es una herramienta de gran utilidad, y es la razón final del uso de *clock*.

6.3. Programa de pruebas

Para medir el rendimiento del código escrito se han diseñado 2 programas: una versión inicial más reducida que compara el tiempo de ejecución de *Expression Templates*, utilizada principalmente en las fases iniciales del desarrollo, y una segunda versión diseñada para probar el rendimiento en un caso específico, pero de una manera mucho más exhaustiva.

Primer programa
La primera versión tiene como objetivo comparar el rendimiento de *Expression Templates* frente al Grafo Dirigido Acíclico ante diversos casos de prueba.
Esta versión se ha diseñado de forma que el programa crea una expresión, inicializa los datos con los que opera de manera aleatoria, evalúa la expresión utilizando ambas versiones, y para cada una, mide el tiempo que tarda en hacerlo.
Puesto que el tiempo que se tarda en realizar la operación es demasiado corto como para analizar las diferencias de una forma precisa, el programa permite incluir un número de repeticiones que se repite cada operación antes de medir el tiempo.

El objetivo de esta versión es analizar el rendimiento del Grafo Dirigido Acíclico ante el máximo número de casos posible, permitiendo obtener conclusiones fiables sobre cómo se comportará en situaciones reales.

Segundo programa
En las fases finales del proyecto, se ha diseñado un segundo programa orientado a analizar el rendimiento del programa en profundidad para una expresión que realiza operaciones de gradiente, una operación que consume mucho tiempo, susceptible a ser afectada por optimizaciones.
Este programa, además de comparar el rendimiento del Grafo Dirigido Acíclico frente a *Expression Templates*, también evalúa la expresión utilizando otros métodos:

- **Expression Templates**
- **Acyclic Directed Graph**
- **Acyclic Directed Graph - By hand**. Una versión del Grafo Dirigido Acíclico, diseñada a mano para el caso concreto, que elimina el gasto de tiempo que requiere el algoritmo normal
- **Expression Templates - By hand**. Una versión de *Expression Templates* diseñada a mano para el caso de prueba.
- **Expression Templates - By hand - cached**. Una versión diseñada a mano que además cachea los resultados intermedios
- **Original**. La versión que se empleaba antes de utilizar *Expression Templates*, que emplea campos intermedios temporales
- **Original - cached**. La versión original, pero reutilizando los valores intermedios

Para cada una de estas técnicas, el programa evalúa la expresión y mide el tiempo que ha tardado en hacerlo, y permite que se exporten a una hoja de cálculo.
El objetivo de esta versión es, para la operación gradiente, una operación realista y muy costosa en tiempo, analizar cómo se comporta el programa evaluándola con todas las técnicas existentes utilizando datos obtenidos de un caso real.

6.4. Resultados obtenidos

6.4.1. Expresiones Simples

Para probar la viabilidad de las simplificaciones de expresiones 1D se ha diseñado un programa que crea expresiones con componentes 1D, y mide el tiempo que conlleva evaluarlas.
Utilizando este programa se han realizado múltiples ejecuciones sin utilizar la optimización, para después repetirlas en condiciones similares aplicando la optimización 1D.

Los resultados obtenidos son los siguientes (resultados en segundos, midiendo tiempo sobre 1000 evaluaciones de la expresión, media de varias ejecuciones):

Expresión	E.T.	Simplificado	Mejora media
$2 * u * PI * r$	1,3156	1,3117	1,0011
$u1 + u2 * l + a/b * c$	1,5763	1.5749	1.0000

Tabla 6.1: Tiempos Simplificación expresiones simples

Las expresiones de la tabla 6.1 utilizan los siguientes elementos:

- *u*, *u1* y *u2* son campos uniformes, que almacenen un mismo valor en todos sus elementos, y es 1D.
- *2*, *PI* y *r* son constantes numéricas, que se construyen como expresiones literales, y son 1D.
- *a*, *b* y *c* son campos escalares, que se construyen como *referenceExpression*, y no son 1D.

Contrario a las expectativas, los resultados obtenidos tras los experimentos no evidencian mejora de rendimiento alguna. Los resultados obtenidos muestran que el tiempo de evaluación de las expresiones es prácticamente el mismo en ambos enfoques.
Una posible explicación para este resultado es que el compilador ya aplica estas optimizaciones de forma nativa, especialmente en compiladores modernos con capacidad de hacer análisis extensivos de las expresiones. Esto provoca que las optimizaciones incluidas a mano no tengan un impacto observable en el rendimiento.

A pesar de no ser los esperados inicialmente, estos resultados no se pueden considerar una pérdida de tiempo, pues resaltan la importancia de realizar pruebas empíricas que validen experimentalmente las optimizaciones propuestas. Adicionalmente, remarcan la importancia de comprender el funcionamiento del compilador sobre el que se trabaja, ya que en ciertas ocasiones las optimizaciones manuales pueden ser redundantes e innecesarias.

6.4.2. Polimorfismo Dinámico

El rendimiento de esta versión del algoritmo se ha probado utilizando el programa de pruebas 1. Para ello, se han seleccionado varias expresiones con operaciones de diversa dificultad, y se han realizado mediciones extensivas sobre ellas. Los resultados obtenidos se ven reflejados en la tabla 6.2, que utiliza las siguientes expresiones:

- Expresión 1: *(a + b) * (a + b)*, una operación muy simple

- Expresión 2: *(-b + sqrt(pow2(b) - 4*a*c)) / (2*a)*, una expresión más compleja, pero que no repite operaciones
- Expresión 3: *vf1 * f(x)-((vf2 + f(x))) * f(x)*, una expresión que contiene la operación *f(x)*, que es extremadamente lenta, hasta el punto de no presentar un uso realista.

El apéndice A incluye una tabla completa con todas las mediciones realizadas para esta versión del algoritmo.

Expresión	Iteraciones	E.T.	ADG	Ratio de mejora
Expresión 1	100	1,7856	14,9770	0,1192
Expresión 2	100	31,2657	82.1405	0,3806
Expresión 3	10	15,9143	5,1506	3,0897

Tabla 6.2: Tiempos Polimorfismo Dinámico

Se han realizado experimentos con expresiones de complejidad creciente con el objetivo de evaluar exhaustivamente la eficacia del algoritmo en situaciones de todo tipo. Se ha demostrado que el algoritmo, teóricamente, cumple su función y reduce el número de operaciones que se realizan. Esto se puede observar analizando el ratio de mejora de la expresión compleja, que resulta en una mejora proporcional al número de veces que aparece la subexpresión común. Sin embargo, en casos realistas y especialmente en expresiones con operaciones simples, utilizar el algoritmo resulta en tiempos mucho mayores que Expression Templates.

A pesar de que utilizar el algoritmo resulta en menos operaciones referentes a la propia expresión, la implementación provoca que internamente se realicen muchas operaciones adicionales.
Esta implementación depende en gran medida del uso del polimorfismo dinámico. Esto provoca que en todas las llamadas a un método de un objeto polimórfico el programa deba buscar en una tabla[21] qué subclase utilizar. Por esta razón, cuantos más elementos tenga el grafo, más búsquedas debe hacer el programa, y más lenta se vuelve la evaluación de la expresión. La expresión compleja tiene operaciones (op1) que tardan más tiempo en procesarse que el tiempo que añade el *Dynamic Dispatching*, razón por la cual obtiene mejores tiempos que Expression Templates.

Estos resultados remarcan la importancia de tener en cuenta todos los factores que afectan al rendimiento cuando se trabaja en entornos de Computación de Altas Prestaciones, donde cualquier sobrecarga puede suponer una gran diferencia en el rendimiento final de un programa.
Por ello, se ha preparado la segunda implementación, que busca reducir este problema y mejorar el rendimiento.

6.4.3. Polimorfismo Estático

De manera similar a la implementación anterior, se han realizado pruebas exhaustivas sobre expresiones cuyas operaciones varían de complejidad con el objetivo de recrear el mayor número de casos posible. Debido a que esta implementación permite modificar si los elementos del grafo cachean o no los resultados intermedios, se han realizado mediciones utilizando ambos modos de funcionamiento. Algunos de los tiempos obtenidos utilizando el programa de pruebas 1 se pueden observar en la tabla 6.3. Esta tabla incluye las mismas expresiones que la tabla 6.2, excepto la expresión 3, que se ha sustituido por la siguiente expresión:

- Expresión 4: *(grad(sf1, mesh) * grad(sf1, mesh)) * sf1*, esta expresión presenta un caso más realista, con 2 operaciones gradiente.

La versión completa de esta tabla está disponible en el apéndice B.

Expresión	Iteraciones	Algoritmo	Tiempo medio	Ratio de mejora
Expresión 1	100	E.T	1,8284	1,0
		ADG	4,2023	0,4350
		ADG - No cache	1,6007	1,1422
Expresión 2	100	E.T	2,6985	1,0
		ADG	10,6945	0,2523
		ADG - No cache	1,9929	1,3540
Expresión 4	10	E.T	23,2616	1,0
		ADG	16,5914	1.4020
		ADG - No cache	31.0848	0.7483

Tabla 6.3: Tiempos Polimorfismo Estático

Se puede observar que en expresiones cuyas operaciones tienen una complejidad computacional reducida, aplicar el algoritmo sigue resultando en tiempos más lentos que Expression Templates. Sin embargo, las expresiones que suponen más carga computacional sí presentan una reducción significativa de tiempo cuando se aplica el algoritmo.
Este experimento demuestra la capacidad del algoritmo de evitar operaciones innecesarias y reducir el tiempo de ejecución, no solo en situaciones teóricas, sino también en casos de uso realistas.

Inicialmente, se planteó como hipótesis el hecho de que almacenar los resultados intermedios podría no ser efectivo para operaciones simples como sumas y restas. Esta hipótesis se ve demostrada en la tabla anterior, donde se puede ver que en operaciones simples, el tiempo de ejecución obtenido recalculando los resultados en todas las llamadas es menor que el tiempo obtenido almacenándolos.

Contrario a las expectativas, este tiempo no solo es más rápido que almacenan-

do resultados intermedios, sino que también supera el rendimiento de *Expression Templates*. Las operaciones que realiza, teóricamente, son las mismas que *Expression Templates*, sin embargo, el tiempo de ejecución es menor.
Una posible explicación para este suceso es que, modificando la estructura de los datos, se consigue que el compilador aplique ciertas optimizaciones que no aplica al árbol de *Expression Templates*, logrando tiempos de ejecución ligeramente inferiores.

6.4.4. Caso de Uso Realista

Una vez realizadas las pruebas de rendimiento teóricas sobre diversos tipos de expresión, se ha realizado una comparativa de rendimiento en profundidad sobre un caso de uso realista. Para ello, se ha hecho uso del programa de pruebas 2.

Para este programa, se ha utilizado una expresión que realiza múltiples veces el gradiente de un campo, una operación que supone mucho cómputo, y, por lo tanto, candidata a ser optimizada.
Esta expresión se ha evaluado utilizando múltiples técnicas con el objetivo de hacer un análisis profundo de cómo alterando la técnica que se utiliza para resolver un problema altera el rendimiento del programa.
Realizando 100 iteraciones de la evaluación sobre campos de diversos tamaños se obtienen los siguientes tiempos: La figura 6.1 es una representación visual de los

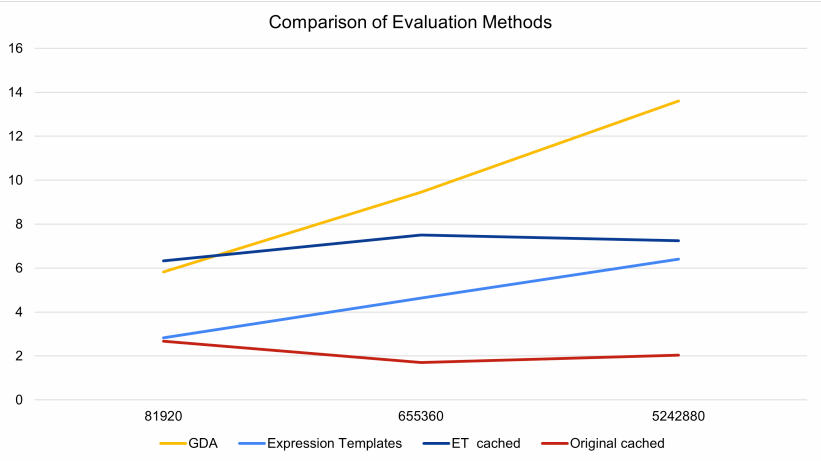

Figura 6.1: Mejora de rendimiento relativa al uso de temporales

datos obtenidos con este programa, que están disponibles en el apéndice C.

En la figura 6.1 se observa la mejora relativa que ofrecen varias técnicas frente al uso de temporales para evaluar una expresión. Cuando se reutilizan los resultados temporales se aprecia una cierta mejora, pues no solo se evitan cálculos innecesarios, también se evitan reservas adicionales de memoria. Sin embargo, la mejora no varía en función del tamaño de los datos de entrada, y no es atractiva

en comparación con otras técnicas analizadas. Esta opción no siempre se utiliza, pues mantener los resultados temporales implica mantener ocupada una gran cantidad de memoria, lo cual puede ser un factor limitante en algunos casos.

Expression Templates supone el siguiente salto de rendimiento, con una mejora que crece de 3 a 5 veces más rápido que el uso de temporales conforme crece el tamaño de los datos de entrada. Esta mejora, añadida al uso reducido de memoria que supone esta técnica, hacen que sea muy atractiva para este tipo de programas, pero se puede observar que tiene un margen de mejora frente a otras técnicas.
Si se calcula el resultado de la operación intermedia y se almacena como un temporal, *Expression Templates* mejora su rendimiento ligeramente, aunque tiende a converger cuando el tamaño de los datos aumenta. Esta opción resulta interesante por su rendimiento ligeramente mayor, aunque implica consumos de memoria elevados y no siempre es posible hacer esta separación de las operaciones.

Sin duda alguna, la mejor opción es el uso del algoritmo de Grafo Dirigido Acíclico, que para un tamaño de los datos relativamente menor tiene un rendimiento similar a otras técnicas, pero conforme crecen los datos, su rendimiento se dispara, llegando a ser 13 veces más rápido que el uso de temporales. Adicionalmente, esta técnica tiene la ventaja de proporcionar consumos de memoria menores, pues utiliza como base el método de *Expression Templates*

Estos resultados evidencian la importancia de analizar el problema que se quiere resolver en profundidad para poder elegir la técnica más adecuada, no solo al problema, sino a los recursos disponibles para resolverlo.

7

Conclusiones y trabajos futuros

El desarrollo del algoritmo del Grafo Dirigido Acíclico ha resultado ser una solución efectiva para mejorar el rendimiento no solo de *OpenFOAM*, sino de cualquier programa basado en el uso de *Expression Templates*.

A pesar de que no siempre sea adecuado almacenar resultados intermedios, la modificación en la estructura de las expresiones ha logrado reducir el tiempo de evaluación de expresiones, tanto complejas como sencillas. De esta forma, los resultados obtenidos han resaltado el hecho de que, incluso en ámbitos maduros como *Expression Templates*, que existe hace más de 20 años, merece la pena realizar investigaciones.

Adicionalmente, se ha logrado desarrollar el algoritmo como una pieza única, requiriendo modificaciones mínimas al proyecto base, permitiendo así una integración sencilla y rápida en cualquier proyecto existente, y actuando de manera transparente al programador.

Si bien no se ha logrado hacer un análisis completo de la expresión en tiempo de compilación, se ha reducido al mínimo el coste que supone este análisis en tiempo de ejecución, pudiendo resultar en un sobrecoste despreciable si se ubica adecuadamente en el código.

Se ha verificado que el compilador identifica correctamente las expresiones sencillas y las evalúa adecuadamente, por lo que no es necesario realizar este proceso manualmente, permitiendo reducir así la complejidad del código.

7.1. Mejoras a futuro

Hay ciertas mejoras que se pueden implementar de cara al futuro con el objetivo de mejorar los aspectos en los que el proyecto presenta carencias.

Se podría mejorar la integración con un proyecto si se elimina el requisito de no modificación del código base. Esto podría hacer el flujo de ejecución y la estructura del código más intuitiva, mejorando la mantenibilidad del sistema.

La sección 5.3.3, en la que se mencionan las dificultades de construir el grafo en tiempo de compilación se podría replantear haciendo uso de *SFINAE*. Para ello, se podría diseñar una estructura que, mediante especializaciones condicionales de una clase, permitiera que el propio compilador construya de manera automática el grafo.
Sin embargo, realizar esta implementación requiere un conocimiento muy avanzado de técnicas de metaprogramación y cambios muy extensos al código, dificultando su integración con el proyecto en su estado actual, y se puede argumentar que comprometería su mantenibilidad.

El mayor potencial de este proyecto de cara al futuro es la capacidad de, partiendo de *Expression Templates*, permitir al programador modificar el orden de evaluación de una expresión. El algoritmo se podría modificar para, en lugar de evaluar una única expresión, realizar la evaluación simultánea de múltiples expresiones como un único grafo, evaluando los nodos comunes a todas las expresiones una única vez. Esta mejora supondría mejoras considerables en el rendimiento sin incurrir en consumos elevados de memoria, algo nunca antes posible.
Implementar esta mejora requeriría cambios muy significativos en la construcción del grafo y, especialmente, en las funciones de evaluación de las expresiones, lo cual aumentaría considerablemente la complejidad general del código y dificultaría mucho su integración en proyectos actuales, que requerirían un proceso de adaptación a este sistema.

Tanto los resultados obtenidos, como la capacidad de evolucionar en el futuro, permiten concluir que este trabajo ha sido un éxito, habiendo cumplido la mayoría de sus objetivos iniciales, y contando con capacidad para mejorar en el futuro.

Bibliografía

[1] J. Hess and A. Smith, "Calculation of potential flow about arbitrary bodies," *Progress in Aerospace Sciences*, vol. 8, pp. 1–138, 1967. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0376042167900036

[2] Y. Toparlar, B. Blocken, B. Maiheu, and G. van Heijst, "A review on the cfd analysis of urban microclimate," *Renewable and Sustainable Energy Reviews*, vol. 80, pp. 1613–1640, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364032117308924

[3] B. Xia and D.-W. Sun, "Applications of cfd in the food industry: a review," 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169901001776

[4] "Ansys scales to reduce cfd simulation time from weeks to one day." [Online]. Available: https://www.ansys.com/blog/ansys-scales-to-reduce-cfd-simulation-time-from-weeks-to-one-day

[5] A. Patil, U. C. Krishnan Paranjothi, and C. García-Sánchez, "Gensdf: An mpi-fortran based signed-distance-field generator for computational fluid dynamics applications," *SoftwareX*, vol. 30, p. 102117, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711025000846

[6] "vector." [Online]. Available: https://en.cppreference.com/w/cpp/container/vector

[7] I. Popov and D. Astakhov, "Implementing memory locality optimizations in openfoam based code," 2023.

[8] H. Mahmoodi Darian, "An efficient method for defining multivariate functions using expression templates for arrays in c++ and cuda," *Journal of Modeling in Engineering*, vol. 16, no. 55, pp. 113–131, 2018.

[9] M. Sagebaum, T. Albring, and N. R. Gauger, "Expression templates for primal value taping in the reverse mode of algorithmic differentiation," *Optimization Methods and Software*, vol. 33, no. 4-6, pp. 1207–1231, 2018.

[10] "Autodesk cfd." [Online]. Available: https://www.autodesk.com/es/products/cfd/overview

[11] "Ansys fluent." [Online]. Available: https://www.ansys.com/products/fluids/ansys-fluent

[12] "openfoam." [Online]. Available: https://www.openfoam.com/

[13] ENGYS, "Helyx core, cfd software," https://engys.com/helyx, 2024.

[14] IBM, "operator overload," 2021. [Online]. Available: https://www.ibm.com/docs/en/zos/2.4.0?topic=only-overloading-operators-c

[15] A. Fluent, "New use cases available with the latest version of the fluent gpu solver," 2024. [Online]. Available: https://www.ansys.com/blog/unleashing-full-power-gpus-ansys-fluent-software-part-3

[16] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.

[17] “Openmpi.” [Online]. Available: https://www.open-mpi.org/

[18] “make.” [Online]. Available: https://www.gnu.org/software/make/

[19] “ninja.” [Online]. Available: https://ninja-build.org/

[20] A. V. Aho, R. Sethi, and J. D. Ullman, *Compiladores: principios, técnicas y herramientas*. Pearson Educación, 1990, ch. 6.1, pp. 359–373.

[21] G. Aigner and U. Hölzle, “Eliminating virtual function calls in c++ programs,” pp. 142–166, 2006. [Online]. Available: https://link.springer.com/chapter/10.1007/bfb0053060