Universidad Carlos III de Madrid
www.uc3m.es

Grado en Ingeniería Informática

2015-2016

Trabajo Fin de Grado

Eliminación de variables globales en aplicaciones C++

Rubén García Sánchez

Tutor

José Daniel García Sánchez

Agradecimientos

A mis padres, sobre todo por aguantarme en los momentos de estrés, pero también por ofrecerme todo lo que está en su mano, apoyarme en mis decisiones y ofrecerme la posibilidad de estudiar lo que quería, a pesar de no entender mucho lo que hago. Gracias por todo.

A mi tutor José Daniel, por confiar en mi desde el principio, por guiarme y ayudarme en este camino.

A David, por facilitarme la iniciación en algo nuevo para mí y por ayudarme en todo cuanto podía.

A mi novia, por aguantarme y apoyarme en los peores momentos, por estar ahí siempre que lo necesito y por mucho más. Gracias por todo.

A mi hermana y a su novio por interesarse y ayudarme a desconectar.

A mis amigos, por ayudarme desconectar de todo y por interesarse.

Índice de contenido

Índice de ilustraciones

Índice de Tablas

1. Introducción

1.1 Motivación

Las variables globales son uno de los pilares en los que se basa la programación y están presentes en la inmensa mayoría de lenguajes existentes. Una variable global es una variable a la que se puede acceder desde cualquier parte del código, ya que como su nombre indica, tiene un ámbito global.

Las variables globales son muy cómodas de utilizar y ahorran al programador el tener que escribir código, por lo que se programa más rápidamente. Cuando se hace uso de una variable global, normalmente es porque va a ser accedida por varias funciones. Al declararse en un ámbito global, las funciones del programa pueden acceder a la variable global sin tener que pasarla como parámetro a dichas funciones. Esta y no otra es la razón fundamental por la que se hace uso de las variables globales, dejando aparte otros temas como la concurrencia o el rendimiento, los cuales no son objetivos de este proyecto.

A pesar de esta más que notable comodidad que nos ofrecen las variables globales, un uso excesivo y sin control de ellas no es recomendable. En algunos lenguajes como C++, se desaconseja su uso. Algunas de las razones por las que se desaconseja su uso son las siguientes:

- Mayor dificultad para entender e interpretar el código.
- Mayor dificultad para depurar el código.
- Existe la posibilidad de introducir errores con declaraciones locales de la misma variable.
- En código concurrente, requieren de la implementación de técnicas de exclusión mutua entre hilos.
- Ocuparán espacio hasta el final de la ejecución del programa, momento en el que son destruidas.

Es evidente que las variables globales no son del todo buenas cuando no son estrictamente necesarias, pero, aun así, es muy frecuente un uso abusivo entre los programadores.

Cambiar la forma de programar de la gente no es fácil. El programador se rige normalmente por la comodidad y por la facilidad. Por ejemplo, es fácil que un programador de C++ utilice la palabra clave *auto* introducida en el estándar de 2011 ya que no tiene que pensar en el tipo de dato que desea y el compilador le va a resolver la tarea. Dejar de hacer uso de las variables globales no es cómodo ni fácil para los programadores porque es algo que llevan usando desde que aprendieron a programar.

Para conseguir reducir esta práctica, es necesario ofrecerle al programador algo cómodo, fácil y que no le quite tiempo. Por esta razón nos planteamos el desarrollar una herramienta de *refactoring* que realice de forma automática esta eliminación de variables globales.

1.2 Objetivos

El objetivo fundamental de este trabajo es desarrollar una herramienta que sea capaz de transformar las variables globales de un programa secuencial en variables locales declaradas en el *main* del programa, con las modificaciones correspondientes en los parámetros de las funciones que hagan uso de ellas. Todo ello para programas escritos en el lenguaje de programación C++.

La herramienta, además de ser un eliminador de variables globales, también será capaz de detectar si las variables existentes no declaradas como constantes pueden ser declaradas como constantes. En caso de que una variable global pueda ser declarada como constante, se transformará a una constante.

Para desarrollar la herramienta haremos uso de LibTooling, una librería basada en el compilador Clang que nos permitirá analizar un fichero de código escrito en C++, obtener información relevante del mismo y modificarlo. La herramienta se desarrollará también con el lenguaje de programación C++.

Como objetivo de funcionamiento, hemos elegido 2 benchmarks pertenecientes a la suite de benchmarks Parsec. Estos dos programas nos servirán para probar el funcionamiento de la herramienta y tener una aplicación de la misma sobre un programa relativamente grande y complejo. El objetivo final es que estos dos programas carezcan de variables globales y el resultado de su ejecución sea el mismo que con el código original.

1.3 Estructura del documento

En este apartado se listan los distintos capítulos que forman este documento y una breve descripción de cada uno de ellos:

- 1. Introducción: describe la motivación y los objetivos de este proyecto.
- 2. Estado del arte: describe de forma teórica, la historia y actualidad de las distintas tecnologías empeladas en el proyecto, así como la relación de este proyecto con otros existentes.
- 3. Análisis del sistema: describe los distintos requisitos que debe tener el sistema y las restricciones del mismo.
- 4. Diseño del sistema: describe la arquitectura adoptada por el sistema y decisiones de diseño.
- 5. Implementación: describe la solución adoptada para desarrollar el sistema y cumplir los requisitos iniciales.
- 6: Pruebas: describe las distintas pruebas a realizar sobre el software para verificar su correcta implementación.
- 7. Conclusiones: contiene un resumen de la satisfacción o no de los requisitos planteados en el análisis, así como algunos posibles trabajos futuros.
- 8. Planificación: describe la planificación inicial y final del proyecto.
- 9. Presupuesto: contiene el cálculo de costes detallado de la realización del proyecto.

1.4 Acrónimos y abreviaturas

SSA (Static Single Assignment): En diseño de compiladores, hace referencia a la propiedad de los datos utilizados por un compilador para representar el código fuente que requiere que cada variable se ha declarado una sola vez y se ha definido antes de usarse.

AST (Abstract Syntax Tree): Representación abstracta en forma de árbol del código fuente de un programa.

```
$ cat test.cc
int f(int x) {
  int result = (x / 42);
  return result;
}

# Clang by default is a frontend for many tools; -Xclang is used to pass
# options directly to the C++ frontend.
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x5aea0d0 <<invalid sloc>>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x5aeab50 <test.cc:1:1, line:4:1> f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 <line:1:7, col:11> x 'int'
  `-CompoundStmt 0x5aead88 <col:14, line:4:1>
    |-DeclStmt 0x5aead10 <line:2:3, col:24>
    | `-VarDecl 0x5aeac10 <col:3, col:23> result 'int'
    |   `-ParenExpr 0x5aeacf0 <col:16, col:23> 'int'
    |     `-BinaryOperator 0x5aeacc8 <col:17, col:21> 'int' '/'
    |       |-ImplicitCastExpr 0x5aeacb0 <col:17> 'int' <LValueToRValue>
    |       | `-DeclRefExpr 0x5aeac68 <col:17> 'int' lvalue ParmVar 0x5aeaa90 'x' 'int'
    |       `-IntegerLiteral 0x5aeac90 <col:21> 'int' 42
    `-ReturnStmt 0x5aead68 <line:3:3, col:10>
      `-ImplicitCastExpr 0x5aead50 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x5aead28 <col:10> 'int' lvalue Var 0x5aeac10 'result' 'int'
```

Ilustración 1 - AST Clang - Fuente: http://clang.llvm.org/docs/IntroductionToTheClangAST.html

ANSI: American National Standards Institute

BSI: British Standards Institute

DIN: German national standards organization

MPI: Message Passing Interface

2. Estado del arte

En este apartado analizaremos la situación actual de todas las tecnologías que se pueden utilizar para desarrollar este proyecto como pueden ser: C++ como lenguaje de programación, herramientas para la refactorización de código escrito en C++ o benchmarks para la evaluación de resultados.

2.1 C++

C++ es un lenguaje de programación de propósito general que nace a partir de C, al cual extiende y mejora de muchas maneras. Algunas de sus características más importantes son:

- Multiparadigma: Procedural (iteradores y algoritmos), funcional, orientado a objetos, programación genérica (plantillas o inferencia de tipos).
- Compilado: que sea compilado quiere decir que el código fuente debe ser procesado por un compilador que genere un programa ejecutable.
- Fuertemente tipado.
- Es portable: el código fuente es portable, no el ejecutable, ya que es el compilador el que se encarga de generar el ejecutable adecuado para la arquitectura sobre la que se está compilando.
- Compatible con C.
- ISO Standard: C++ está estandarizado, oficialmente conocido como ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++.

Historia

C++ nace en 1979 de la mano de Bjarne Stroustrup, con el denominado “C con clases” en el que las características más importantes del lenguaje eran la existencia de las clases, las funciones y los constructores y destructores. Su objetivo era introducir el paradigma de la programación orientada a objetos sobre C, paradigma al cual le vio una gran utilidad mientras desarrollaba tu tesis doctoral [1].

En 1983, el denominado “C con clases” acuñó el término de C++. El gran crecimiento del uso de C++ hizo ver de forma clara Bjarne Stroustrup que tarde o temprano iba a ser necesaria una estandarización del lenguaje. Durante los siguientes años Stroustrup se dedicó a contactar con las empresas desarrolladoras de compiladores para C++ y con las empresas que más uso hacían del lenguaje para llegar a acuerdos. En 1989 se creó el primer comité de estandarización, el comité X3J16 de la ANSI que más tarde, en 1991, pasaría a formar parte de un nuevo comité de la ISO llamado WG21 [1].

En el año 1998 se pública el primer estándar de C++ por la ISO en colaboración con diferentes organizaciones de estandarización como ANSI, BSI o DIN [1].

Actualidad

A pesar de la corrección del estándar C++98 que se llevó acabo en el 2003, no se publica un nuevo standard hasta el 2011 con C++11. Las mejoras son numerosas y muy potentes, como dice Stroustrup, C++11 parece un lenguaje nuevo, en el que se escribe código de forma diferente, más corto, simple y más eficiente que antes [2].

El objetivo general de este nuevo estándar según Stroustrup son los siguientes:

- Hacer de C++ un lenguaje mejor para la programación de sistemas y el desarrollo de librerías [3].
- Hacer de C++ un lenguaje cada vez más sencillo de enseñar y aprender [3].

Algunas de las nuevas características presentes en este estándar son las siguientes:

- **auto:** esta nueva palabra clave especifica que el tipo de dato que tendrá la variable será deducido automáticamente a partir de su valor de inicialización. Ejemplo: `auto x = 7;`
- **decltype:** esta palabra clave sirve para declarar una variable con el tipo de dato resultante de una expresión determinada. A diferencia de auto, que se utiliza con una variable que vamos a inicializar, decltype sirve para inferir el tipo de una expresión más que de una variable concreta [4].
- **Range-for:** Range-for hace referencia a una nueva forma de iterar a través de un rango de elementos de forma muy sencilla. Válida para cualquier contenedor estándar, array, o cualquier conjunto de elementos que tengan definido un begin() y un end() [4]. Ejemplo: `for (auto x : v),` en cada iteración del bucle tendremos almacenado en x el siguiente valor del conjunto de elementos v.
- **constexpr:** esta nueva palabra clave ofrece una nueva forma de declarar expresiones constantes de forma más general. Ofrece además una forma de cerciorarse de que una inicialización se está realizando en tiempo de compilación. Es un gran elemento para incrementar el rendimiento de programas [4].
- **Mejoras en los contenedores** [5]**:**
    - Inicialización de contenedores mediante listas.
    - Mejoras en las operaciones de inserción.
    - Introducción del operador *move.*

Como ya se ha comentado, estas son solo una muestra muy pequeña de las nuevas características del estándar C++11, existen muchísimas mejoras más referentes a las clases, templates, concurrencia o contenedores.

La actualización más reciente del estándar es la del año 2014 y conocida como C++14. Esta revisión se considera como una pequeña extensión del estándar C++11 en la que se corrigen bugs y se realizan pequeñas mejoras [6].

Futuro

Desde el 2014, el comité sigue trabajando en mejoras de diferentes ámbitos como librerías de bajo nivel, paralelismo, concurrencia, etc. Muchas de estas mejoras están previstas que formen parte de la próxima gran revisión del estándar, fechada para 2017 [7].

2.2 LLVM

Antes de comenzar a hablar de Clang, es necesario hablar sobre LLVM. LLVM es una tecnología compuesta por una colección de herramientas modulares y reutilizables para la compilación.

LLVM surge como un proyecto de investigación en la Universidad de Illinois, con el objetivo de conseguir una estrategia de compilación basada en SSA capaz de soportar la compilación estática y dinámica de programas escritos en diferentes lenguajes. En la actualidad LLVM está compuesto por un conjunto de proyectos diferentes y es usado por multitud de herramientas comerciales y de código abierto como lo es Clang [8].

Algunos de los principales proyectos de los que se compone LLVM en la actualidad son los siguientes:

- **LLVM Core:** librerías que permiten la optimización y generación de código. Para múltiples CPUs [8].
- **Clang:** es un compilador basado en LLVM para C/C++/Objective-C del que se hablará más adelante.
- **Dragonegg:** integra LLVM Core con *parsers* de GCC. Permite a LLVM compilar ficheros escritos en lenguajes como Ada, Fortran u otros lenguajes soportados por GCC [8].
- **LLDB:** herramienta nativa para el depurado creada con librerías de LLVM y Clang [8].
- **Libc++:** implementación de gran rendimiento de la librería Estándar de C++, incluyendo soporte para C++11 [8].
- **vmkit:** implementación de las máquinas virtuales de Java y .NET basada en LLVM [8].
- **polly:** suite que implementa optimizaciones para la caché basadas en localidad espacial, así como paralelización y vectorización haciendo uso de un modelo poliédrico [8].

Como se puede observar, el uso de LLVM está muy extendido y es muy variado, no es de extrañar que sea la base sobre la que se cimienta Clang.

2.3 Clang

Clang es un front-end de C/C++/Objective-C para el compilador LLVM. Clang surge por la necesidad de tener un compilador que realice un mejor diagnóstico que los existentes, que sea capaz de integrarse con distintos entornos de desarrollo, que sea compatible con productos comerciales y que sea fácil de desarrollar y mantener [9]. Algunas de las características más importantes de Clang, según sus creadores, son:

- Rápido y con un bajo uso de memoria [10].
- Ofrece una gran cantidad de feedback al usuario en sus diagnósticos tras la compilación [10].
- Sus librerías están estructuradas de forma modular [10].
- Integración con IDEs [10].
- Soporte para el estándar C++11 [10].
- Código simple y fácil de entender [10].
- Proporciona herramientas para analizar el código fuente en tiempo de compilación.
- Proporciona herramientas para la transformación del código fuente en tiempo de compilación.

En la actualidad, Clang, además de ser un compilador, es la solución más utilizada para desarrollar herramientas que analizan y tratan el código fuente en tiempo de compilación, gracias a las librerías que proporciona. Además de ofrecer estas librerías para la creación de herramientas, Clang ofrece la posibilidad de integrar en futuras versiones aquellas herramientas desarrolladas por gente de la comunidad que cumplan con ciertos criterios que imponen.

Existen varias interfaces diferentes para desarrollar herramientas en base a las necesidades que tenga el desarrollador. Existen 3 interfaces distintas:

- **LibClang:** es una interfaz escrita en C con un alto nivel de abstracción. Recomendada cuando no se va a hacer uso de las características de C++ o cuando no se requiere de un control total del AST. Es la interfaz recomendada en caso de duda. [11]

- **Clang Plugins:** permite realizar acciones adicionales en el AST como parte del proceso de compilación. Los plugins son librerías dinámicas que se cargan en tiempo de ejecución por el compilador. Son dependientes del entorno para el que se crean por lo que no se recomiendan para herramientas independientes. [11]

- **LibTooling:** es una interfaz escrita en C++ cuyo objetivo es el de facilitar el desarrollo de herramientas independientes y su integración con los servicios de clang. Su uso se recomienda cuando se requiere de un control completo sobre el AST o cuando se quiere ejecutar la herramienta sobre un determinado fichero o conjunto de ficheros. [11]

2.4 Clang tools

Clang tools es un conjunto de herramientas de línea de comandos diseñadas para desarrolladores de C++ que hacen uso de Clang como compilador. Las herramientas están enfocadas en ayudar al desarrollador y las hay de diversas utilidades [12].

Las herramientas se organizan en 2 grupos diferentes: las más básicas y fundamentales se encuentran en el proyecto principal de clang y el resto se encuentran en otro proyecto de manera que quien no las necesite no esté obligado a descargarlas.

La característica principal de estas herramientas es que se desarrollan haciendo uso de las interfaces comentadas en el punto anterior y que son las que se usarán para desarrollar este proyecto.

Core Clang Tools

Estas son las herramientas que están en el repositorio principal. Su objetivo es el de permitir al usuario usar y probar funcionalidades específicas de Clang [12].

Clang-check

Esta herramienta hace uso de la interfaz de libtooling y del compilador de Clang para analizar el código fuente de los ficheros de forma rápida y a través de la línea de comandos, proporcionando comprobación de errores y mostrando el AST del código fuente [13].

```
$ cat <<EOF > snippet.cc
> void f() {
>   int a = 0
> }
> EOF
$ ~/clang/build/bin/clang-check snippet.cc -ast-dump --
Processing: /Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc.
/Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc:2:12: error: expected ';' at end of
      declaration
  int a = 0
           ^
           ;
(TranslationUnitDecl 0x7ff3a3029ed0 <<invalid sloc>>
  (TypedefDecl 0x7ff3a302a410 <<invalid sloc>> __int128_t '__int128')
  (TypedefDecl 0x7ff3a302a470 <<invalid sloc>> __uint128_t 'unsigned __int128')
  (TypedefDecl 0x7ff3a302a830 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]')
  (FunctionDecl 0x7ff3a302a8d0 </Users/danieljasper/clang/llvm/tools/clang/docs/snippet.cc:1:1, line:3:1> f 'void (void)'
    (CompoundStmt 0x7ff3a302aa10 <line:1:10, line:3:1>
      (DeclStmt 0x7ff3a302a9f8 <line:2:3, line:3:1>
        (VarDecl 0x7ff3a302a980 <line:2:3, col:11> a 'int'
          (IntegerLiteral 0x7ff3a302a9d8 <col:11> 'int' 0))))))
1 error generated.
Error while processing snippet.cc.
```

Ilustración 2- Ejemplo clang-check - Fuente: http://clang.llvm.org/docs/ClangCheck.html

Clang-format

Clang-format es una librería y una herramienta independiente cuyo objetivo es el de dar formato a los ficheros de código escritos en C++ en base a una serie de estilos configurables. para ello hace uso de la interfaz proporcionada por la clase Lexer que transforma los ficheros en una secuencia de tokens para poder trabajar con ellos [12].

Extra Clang Tools

El resto de herramientas que no se encuentran en el repositorio principal se encuentran en este repositorio extra.

Clang-tidy

Clang tidy es una herramienta, escrita con la interfaz de libtooling, que analiza el código fuente de forma estática para buscar y corregir errores típicos de programación. Está implementada de forma modular y cuenta con un gran número de pequeñas herramientas diferentes que realizan una sola comprobación y su correspondiente corrección [14].

Las distintas herramientas existentes vienen clasificadas en distintos grupos. A continuación, se mostrará esta clasificación y se hará hincapié en aquellos grupos que son más relevantes para este proyecto. La clasificación, encontrada en [14], es la siguiente:

- Herramientas que comprueban el código referente a convenciones de LLVM.
- Herramientas que comprueban el código referente a convenciones de Google.
- Herramientas que comprueban el código referente al uso de C++11 (Clang modernizer). Este grupo es uno de los más relevantes para nuestro proyecto ya que busca elementos concretos de código y los reemplaza por código equivalente escrito en el estándar C++11. Algunos ejemplos de comprobaciones pertenecientes a este grupo son:
    - **modernize-loop-convert:** esta herramienta busca bucles del tipo for(...; ...; ...) para transformarlos por los nuevos bucles range-for incorporados en el estándar C++11 [15].
    - **modernize-use-nullptr**: esta herramienta localiza y convierte punteros constantes a *null* por la nueva palabra clave incorporada en el estándar C++11 *nullptr [16].*

- **modernize-use-auto:** herramienta que transforma la declaración de variables para sustituir el tipo especificado por la palabra clave *auto [*17*].* Por ejemplo:

```
std::vector<int>::iterator I = my_container.begin();

// transforms to:

auto I = my_container.begin();
```

Ilustración 3- Ejemplo auto – Fuente: http://clang.llvm.org/extra/clang-tidy/checks/modernize-use-auto.html

- Herramientas relacionadas con la legibilidad del código.
- Herramientas que realizan comprobaciones estáticas con Clang.
- Herramientas relacionadas con la optimización del rendimiento.
- Otras herramientas que no encajan en las anteriores categorías.

La lista completa de comprobaciones se puede encontrar en [18].

Clang tidy ofrece también una interfaz para que los desarrolladores puedan crear sus propias herramientas para realizar comprobaciones y transformar el código [14].

2.5 Antecedentes en eliminación de variables globales

Poco a poco hemos ido avanzando desde lo más general a lo más concreto de las tecnologías de las que se va a hacer uso. En este punto se describen un par de aplicaciones más concretas y relacionadas con el objetivo de este proyecto, la eliminación de variables globales. En estas aplicaciones se hace uso de las tecnologías que serán usadas para realizar el proyecto.

Existen muy pocos estudios relacionados con el estudio que pretendemos llevar a cabo y solo existe uno que se asemeje a lo que nosotros pretendemos hacer. Por un lado, tenemos el artículo [19], en el que se analiza el efecto que producen las variables globales en un ámbito específico. Este estudio sólo se centra en variables globales que producen dependencias entre los componentes de un programa por lo que no se analiza de forma genérica el impacto de las variables globales como nosotros pretendemos.

Por otro lado, en el artículo [20], se aproximan un poco más a lo que nosotros pretendemos hacer, sin embargo, su objetivo es conseguir que los programas MPI multihilo sean seguros, es decir, que solo uno de los hilos acceda a los recursos compartidos entre todos, en este caso las variables globales. Para ello plantean 3 soluciones, una de las cuales se aproxima a nuestro planteamiento, reescribir el código transformando las variables globales en locales de forma automatizada. Además, no se realiza una comparativa de rendimiento entre el programa original y el de la transformación como nosotros pretendemos hacer, si no que se comparan las distintas técnicas presentadas entre sí.

Improving performance and maintainability through refactoring in C++11

Este título hace referencia al título de un artículo publicado por Bjarne Stroustrup, creador del lenguaje C++, y José Daniel García Sánchez, profesor de la Universidad Carlos III de Madrid y miembro del comité internacional ISO para la normalización de C++.

El artículo trata sobre el hecho de si aumentar el grado de abstracción de un programa implica perdida de rendimiento. Para ello transformarán la aplicación *fluidanimate* perteneciente a la suite de benchmarks PARSEC, partiendo de una versión de código de bajo nivel escrita en C para llegar a una versión con un mayor nivel de abstracción escrita en C++11 [21].

Dentro de las transformaciones que se realizan sobre el código, está la eliminación de variables globales y constantes. Comentan en [21] que muchos desarrolladores asumen que el uso de variables globales ofrece una ventaja en cuanto a rendimiento se refiere y que por tanto se hace un uso abusivo de ellas independientemente de que esto afecte a la mantenibilidad del código. También afirman que hoy día el acceso a las variables globales puede requerir un mayor número de instrucciones por acceso que las variables locales (almacenadas en la pila), sobre todo cuando se accede a la variable en múltiples ocasiones.

Normalmente, en un programa los distintos tipos de variables globales que podemos encontrar son los siguientes:

1. Constantes con valor conocido en tiempo de compilación.
2. Constantes a partir de una expresión conocida en tiempo de compilación.
3. Constantes definidas con preprocesador.
4. Constantes con valor establecido a principio de programa (p.ej. leído de un fichero o suministrado por el usuario).
5. Variables globales usadas para evitar pasarlas como parámetro.

Para cada tipo de variable global se propone una transformación manual, basada en optimizar el rendimiento. Por ejemplo, las constantes cuyo valor es calculado al inicio del programa, se agrupan todas en una sola clase como constantes públicas no estáticas. De esta forma y haciendo uso de otras transformaciones, los autores consiguen obtener una implementación equivalente a la versión original con un rendimiento mayor [21].

Source-to-Source Refactoring and Elimination of Global Variables in C Programs

Este título hace referencia a un artículo publicado por Hemaiyer Sankaranarayanan y Prasad A. Kulkarni en el *Journal of Software Engineering and Applications.*

Este artículo es el único trabajo encontrado cuya finalidad es la misma que la de este proyecto, eliminar las variables globales de programas. Ellos tienen como objetivo, además de los programas escritos en C++, los programas escritos en C.

Los autores presentan en [22] una herramienta, basada en el compilador clang, para la transformación del código fuente de los programas. La funcionalidad de la herramienta se basa en la eliminación de las variables globales y su consiguiente redefinición y paso por parámetros a las funcionas que hacían uso de ellas. Además de presentar esta herramienta, realizan un estudio del efecto que tienen las transformaciones en lo que a rendimiento se refiere.

Seguidamente, se analizan las ventajas y desventajas de las variables globales y cuentan cómo han desarrollado su herramienta de una forma muy superficial, sin entrar en nada referente a la clase de código desarrollado, solo hacen referencia a las tecnologías utilizadas.

Finalmente, presentan un estudio comparando el rendimiento de las aplicaciones originales utilizadas para medir, con el rendimiento de las mismas aplicaciones tras haber sido transformadas por la herramienta que han desarrollado. Los resultados obtenidos, en cuanto a rendimiento se refiere, no son muy positivos ya que, en la mayoría de los casos, el rendimiento del programa transformado es igual o peor al original, aunque con las ventajas que supone no usar variables globales.

2.6 *Benchmark* PARSEC

Independientemente de que este proyecto no tenga como fin la evaluación del rendimiento del código resultante tras eliminar las variables globales, es necesario realizar pruebas consistentes para comprobar el correcto funcionamiento de la herramienta. Para realizar las pruebas que nos permitan verificar un correcto funcionamiento de la herramienta se va a hacer uso de Parsec benchmark.

PARSEC (Princeton Application Repository for Shared-Memory Computers) es una suite de benchmarks compuesta por programas multithread. La suite, compuesta por 13 programas, se centra en evaluar procesadores multicore que hacen uso de memoria compartida [23]. Las características más representativas de PARSEC [23] son:

- Multithread: sus benchmarks están paralelizados y permiten su ejecución de forma paralela.
- Código novedoso y emergente.
- Diversidad: la selección de benchmarks trata de ser lo más diversa y representativa posible. No se centran en un dominio específico.
- No se centran en computación de altor rendimiento. Hay benchmarks de distintas cargas de trabajo y dominios.
- Su uso fundamental es para investigación, aunque también es válido para medir el rendimiento en las máquinas.

Cada benchmark tiene, por defecto, seis posibles entradas de datos para las ejecuciones. Las diferentes entradas son las siguientes [24]:

- Test: la entrada más pequeña posible para probar el funcionamiento básico del programa.
- Simdev: Similar a la anterior, pero con una carga mayor.
- Simsmall: entrada de datos real con un tiempo medio de ejecución aproximado a 1 segundo.
- Simmedium: entrada de datos real con un tiempo medio de ejecución aproximado a 5 segundos
- Simlarge: entrada de datos real con un tiempo medio de ejecución aproximado a 15 segundos.
- Native: entrada de datos real con un tiempo medio de ejecución aproximado a 15 minutos.

Además de estas entradas por defecto, algunos de los benchmarks vienen con ejecutables para generar otros ficheros de entrada, para que cada desarrollador pueda probar lo que necesite.

Como se ha comentado, los benchmarks incluidos en la suite son 13. Para obtener una ligera idea de que trata cada benchmark vamos a describirlos de forma muy breve [25]:

Blackscholes: este benchmark de Intel calcula precios de forma analítica haciendo uso de la ecuación diferencial parcial Black-Scholes.

Bodytrack: aplicación de computación visual de Intel que realiza un seguimiento de un cuerpo humano con múltiples cámaras a través de una secuencia de imágenes.

Canneal: kernel desarrollado por la universidad de Priceton que calcula el menor coste de enrutamiento para el diseño de un chip.

Dedup: kernel desarrollado por la universidad de Priceton que comprime un conjunto de datos mediante una combinación de compresiones globales y locales denominada deduplicación.

Faceism: aplicación de computación visual de Intel que computa una animación visual realista de un modelo facial simulando las físicas.

Ferret: aplicación desarrollada por la universidad de Priceton basada en la herramienta Ferret que sirve para búsqueda de contenidos por similitud. En el benchmark está implementado para buscar imágenes similares a una dada.

Fluidanimate: aplicación de Intel que simula el comportamiento de un fluido para animaciones.

Freqmine: aplicación que implementa una versión basada en arrays del *método FP-growth (Frequent Pattern-growth)* para *Frequent Itemset Mining (FIMI*).

Streamcluster: kernel desarrollado por la universidad de Priceton que resuelve el problema del clustering online.

Swaptions: aplicación de Intel que hace uso del Heath-Jarrow-Morton (HJM) framework para poner precio a un fichero de Swaptions.

Vips: aplicación basada en el sistema de procesamiento de imágenes VASARI.

x264: aplicación que realiza codificación H.264/AVC (Advanced Video Coding).

Como complemento a estas breves descripciones de cada aplicación, en la siguiente imagen se puede ver una comparativa de las distintas características de cada uno de los benchmarks [24].

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Ilustración 4-parsec características – Fuente http://parsec.cs.princeton.edu/doc/parsec-report.pdf

3. Análisis del sistema

3.1 Detalle de la aplicación

El objetivo de esta herramienta es eliminar las variables globales de programas escritos de C++ modificando el código fuente de los programas. El funcionamiento básico con el que contará la herramienta será el siguiente:

- Identificar variables globales.
- Identificar constantes definidas con macros.
- Clasificar las variables globales en constantes o no constantes.
- Eliminar variables globales y reescribirlas en el main.
- Reescritura de variables que no estaban declaradas como constantes, pero realmente son constantes.
- Modificar las definiciones de las funciones, añadiendo tantos parámetros nuevos a las funciones como variables globales eliminadas se usaban en ella. De la misma forma, añadir los parámetros en las llamadas a estas funciones.
- No modifica el código fuente si detecta errores o incompatibilidades en el código durante la ejecución.

Todo el proceso de transformación se hará con una única herramienta y en una sola ejecución.

Se modificará el fichero fuente pasado por parámetro y todos aquellos ficheros, que estén exactamente en el mismo directorio que el fichero fuente, de los que haga uso. No se crearán copias de los ficheros ni se modificarán copias del mismo, se modificarán los ficheros originales.

El usuario debe cerciorarse de que su código está exento de errores de compilación ya que el resultado de la transformación se puede ver alterado si existen errores de compilación, es decir, se asume que el código que se le pasa a la herramienta compila.

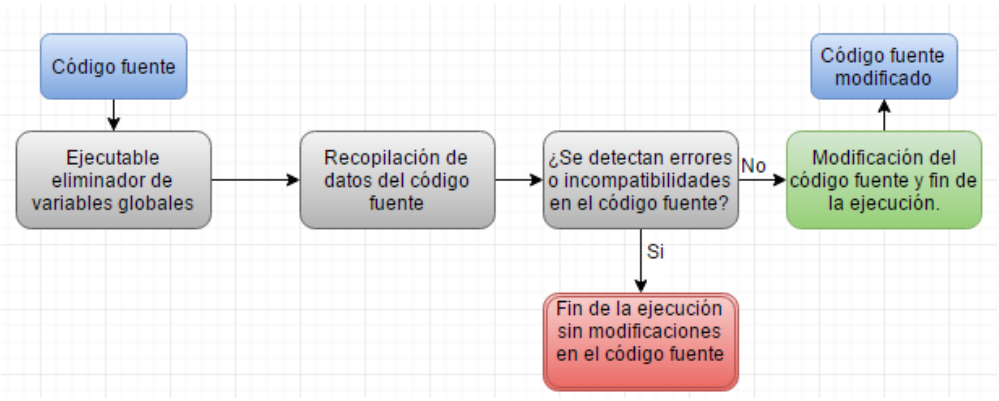

Ilustración 5 - Esquema de funcionamiento

3.2 Requisitos de Usuario

Los requisitos de usuario son la descripción de lo que el cliente quiere o necesita de un software. Estos requisitos se obtienen, normalmente, mediante diversas reuniones con el cliente. En este caso el cliente es nuestro tutor. La recogida de estos requisitos se ha llevado a cabo mediante diferentes reuniones con el tutor.

Vamos a distinguir tres tipos de requisitos de usuario:

- ***Requisitos de capacidad:*** Son aquellos que representan lo que los usuarios necesitan para llevar a cabo un objetivo o resolver un problema.
- ***Requisitos de restricción:*** Son aquellas restricciones impuestas por los usuarios sobre cómo llevar a cabo un objetivo o cómo resolver un problema.
- ***Requisitos inversos:*** Se encargan de describir lo que el sistema no hará. Útiles especialmente para evitar confusiones.

Por otra parte, para definir cada requisito de usuario incluiremos en cada uno los siguientes atributos:

- **Identificador:** Sirve para identificar cada requisito y así facilitar la trazabilidad.

  Identificador seguirá el siguiente formato: RUX-YYY
  - *RU:* Indica que es un requisito de usuario.
  - *X:* Indica el tipo de requisito, y admite los valores:
    - ✓ ***C:*** Indica que es un requisito de tipo Capacidad.
    - ✓ ***R:*** Indica que es un requisito de tipo Restricción.
    - ✓ ***I:*** Indica que es un requisito de tipo Inverso.
  - *YYY:* Número Comprendido entre 001 y 999 que identifica al requisito de usuario dentro de su tipo.
- **Prioridad:** Indica el valor de prioridad para facilitar la planificación del programador.
- **Necesidad:** El nivel de necesidad indicará si el requisito es negociable o no con el cliente.
- **Tipo:** se distingue entre requisitos de capacidad, de restricción o inversos.
- **Descripción:** Reseña dónde se describe el requisito de usuario.

La tabla que muestra la información de cada requisito es una como la siguiente:

IDENTIFICADOR: RUX-YYY
PRIORIDAD:
NECESIDAD:
TIPO:
DESCRIPCIÓN:

Tabla 1 - Ejemplo tabla requisitos de usuario.

Una vez explicada toda la información necesaria, a continuación, se muestran todos los requisitos de usuario obtenidos.

Requisitos de capacidad

IDENTIFICADOR: RUC-01
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Capacidad
DESCRIPCIÓN: La herramienta borra las variables globales que no están definidas como expresiones constantes.

Tabla 2 - RUC-01

IDENTIFICADOR: RUC-02
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Capacidad
DESCRIPCIÓN: La herramienta reescribe las variables globales borradas en el *main* del programa.

Tabla 3 - RUC-02

IDENTIFICADOR: RUC-03
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Capacidad
DESCRIPCIÓN: La herramienta añade los parámetros necesarios a las funciones que hacen uso de las variables globales borradas.

Tabla 4 - RUC-03

IDENTIFICADOR: RUC-04
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Capacidad
DESCRIPCIÓN: La herramienta añade como parámetro de las llamadas a una función las variables globales borradas que se utilicen en dicha función.

Tabla 5 - RUC-04

IDENTIFICADOR: RUC-05
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Capacidad
DESCRIPCIÓN: La herramienta reescribe las variables globales como constexpr o, en su defecto, como const siempre que sea posible.

Tabla 6 - RUC-05

IDENTIFICADOR: RUC-06
PRIORIDAD: Media
NECESIDAD: Obligatoria
TIPO: Capacidad
DESCRIPCIÓN: La herramienta reescribe como expresiones constantes globales (constexpr) las constantes definidas mediante macros.

Tabla 7 - RUC-06

Requisitos inversos

IDENTIFICADOR: RUI-01
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Inverso
DESCRIPCIÓN: La herramienta no borra ni modifica las variables globales definidas con constexpr.

Tabla 8 - RUI-01

IDENTIFICADOR: RUI-02
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Inverso
DESCRIPCIÓN: La herramienta no modifica las variables globales definidas entre macros.

Tabla 9 - RUI-02

IDENTIFICADOR: RUI-03
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Inverso
DESCRIPCIÓN: La herramienta no modifica las variables globales externas (declaradas con extern).
JUSTIFICACIÓN: No suponen un problema.

Tabla 10 - RUI-03

Requisitos de restricción

IDENTIFICADOR: RUR-01
PRIORIDAD: Alta
NECESIDAD: Obligatorio
TIPO: Restricción
DESCRIPCIÓN: La herramienta deberá funcionar en un computador con sistema operativo Linux.

Tabla 11 - RUR-01

IDENTIFICADOR: RUR-02
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Restricción
DESCRIPCIÓN: El sistema operativo hará uso de la versión 3.6 de Clang o superiores.

Tabla 12 - RUR-02

IDENTIFICADOR: RUR-03
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Restricción
DESCRIPCIÓN: La herramienta debe probarse con, al menos, 2 benchmarks de la suite de PARSEC.

Tabla 13 - RUR-03

IDENTIFICADOR: RUR-04
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Restricción
DESCRIPCIÓN: Se hará uso del estándar C++11 para desarrollar la herramienta.

Tabla 14 - RUR-04

3.3 Casos de uso

En este punto se van a especificar los casos de uso gracias a la definición de los Requisitos de Usuario definidos anteriormente. El objetivo de describir los casos de uso es el de describir lo que el usuario puede realizar con la herramienta. En este caso tenemos una herramienta automática que realiza una determinada funcionalidad tras ejecutar por lo que solamente vamos a tener un solo caso de uso.

Cada caso de uso viene definido por los siguientes atributos:

- Identificador: código que identifica unívocamente a cada uno de los casos de uso.
- Nombre: nombre esclarecedor de cada caso de uso.
- Actores: tipo de usuario para el que va dirigido el caso de uso.
- Objetivo: la finalidad que persigue el caso de uso en cuestión.
- Descripción: explicación concisa de cómo el actor del caso interactúa con el sistema
- Precondiciones: condiciones que debe cumplir para poder realizar la operación.
- Postcondiciones: estado en el que queda el sistema tras realizar la operación.
- Condiciones de fallo: posibles fallos durante la operación y la respuesta ante el fallo.

Los casos de uso se definen a continuación:

Identificador	CU-01
Nombre	Ejecución
Actores	Usuario
Objetivo	Eliminar las variables globales de un programa
Descripción	El usuario ejecuta la herramienta pasando por parámetro la ruta completa del fichero de código fuente que contiene el main del programa.
Precondiciones	Tener el ejecutable de la herramienta y las librerías para enlazar.
Postcondiciones	El programa del usuario ha sido modificado por la herramienta.
Condiciones de fallo	Si se detecta algún problema durante la ejecución de la herramienta, el código fuente del usuario no se verá modificado.

Tabla 15 - Caso de uso

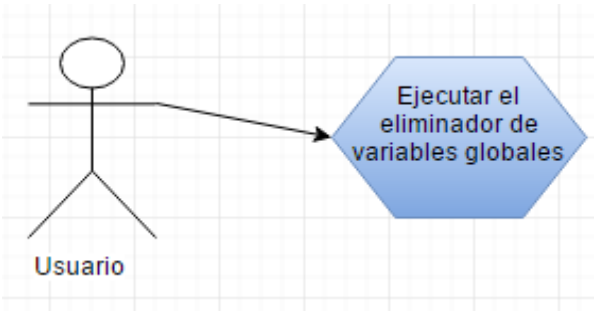

Ilustración 6 - Caso de uso

3.4 Requisitos Software

En este punto se detallarán las características necesarias para el desarrollo de la herramienta. Cada una de las tablas tendrá una serie de información que se explica a continuación:

- **Identificador**: campo utilizado para identificar de forma unívoca un requisito. El identificador tendrá el siguiente formato: XX-YY, donde XX hace referencia al tipo de requisito e YY el número de requisito de ese tipo. XX tomará el valor RF para los requisitos funcionales y RNF para los requisitos no funcionales.
- **Prioridad**: campo que informa de la importancia del requisito. Sus posibles valores son: baja, media o alta.
- **Necesidad**: su implementación puede ser obligatorio u opcional.
- **Tipo**: se distingue entre requisitos funcionales o no funcionales.
    - **Funcionales**: especifican que es lo que tiene que hacer el software.
    - **No funcionales**: especifican características del funcionamiento, restricciones de diseño.
- **Descripción**: explicación del requisito.
- **Justificación**: razonamiento de la existencia de ese requisito.

La tabla que muestra la información de cada requisito es una como la siguiente:

IDENTIFICADOR: XXX
PRIORIDAD:
NECESIDAD:
TIPO:
DESCRIPCIÓN:
JUSTIFICACIÓN:

Tabla 16 - Ejemplo tabla requisitos software

Una vez explicada toda la información necesaria, a continuación, se muestran todos los requisitos software obtenidos.

Requisitos no funcionales

IDENTIFICADOR: RNF-01
PRIORIDAD: Alta
NECESIDAD: Obligatorio
TIPO: No funcional
DESCRIPCIÓN: La herramienta deberá funcionar en un computador con sistema operativo Linux.
JUSTIFICACIÓN: Las herramientas necesarias para el desarrollo y generación de la herramienta funcionan en Linux.

Tabla 17 - RNF-01

IDENTIFICADOR: RNF-02
PRIORIDAD: Alta
NECESIDAD: Obligatorio
TIPO: No funcional
DESCRIPCIÓN: La herramienta se desarrollará en un computador con sistema operativo Linux.
JUSTIFICACIÓN: Las herramientas necesarias para el desarrollo y generación de la herramienta funcionan en Linux.

Tabla 18 - RNF-02

IDENTIFICADOR: RNF-03
PRIORIDAD: Media
NECESIDAD: Opcional
TIPO: No funcional
DESCRIPCIÓN: El sistema operativo Linux requerido en el RNF-02 debe ser un sistema operativo Ubuntu versión 14.04 o superior.
JUSTIFICACIÓN: Mayor compatibilidad de herramientas y el sistema operativo está más actualizado.

Tabla 19 - RNF-03

IDENTIFICADOR: RNF-04
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: No funcional
DESCRIPCIÓN: El sistema operativo hará uso de la versión 3.6 de Clang o superiores.
JUSTIFICACIÓN: No conviene utilizar versiones excesivamente antiguas ya que frecuentemente se corrigen bugs.

Tabla 20 - RNF-04

IDENTIFICADOR: RNF-05
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: No funcional
DESCRIPCIÓN: La herramienta debe probarse con, al menos, 2 benchmarks de la suite de PARSEC.
JUSTIFICACIÓN: Es necesario probar la herramienta con algún ejemplo real de código.

Tabla 21 - RNF-05

IDENTIFICADOR: RNF-06
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: No funcional
DESCRIPCIÓN: El resultado de ejecutar la versión original del benchmark y el resultado de ejecutar la versión modificada por la herramienta debe ser el mismo.
JUSTIFICACIÓN: Solo de esta manera podemos garantizar el correcto funcionamiento de la herramienta desarrollada.

Tabla 22 - RNF-06

IDENTIFICADOR: RNF-07
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: No funcional
DESCRIPCIÓN: Se hará uso del estándar C++11 para desarrollar la herramienta.
JUSTIFICACIÓN: Solo se puede desarrollar en C++ y por ello hacemos uso del estándar C++11.

Tabla 23 - RNF-07

IDENTIFICADOR: RNF-08
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: No funcional
DESCRIPCIÓN: Se hará uso de las librerías/API/interfaz que proporciona Clang con libtooling para desarrollar la herramienta.
JUSTIFICACIÓN: Es la única interfaz que nos permite desarrollar esta herramienta.

Tabla 24 - RNF-08

Requisitos funcionales

IDENTIFICADOR: RF-01
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta detecta variables globales.
JUSTIFICACIÓN: Para poder eliminarlas primero hay que detectarlas.

Tabla 25 - RF-01

IDENTIFICADOR: RF-02
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta detecta constantes globales definidas con macros.
JUSTIFICACIÓN: Es otro tipo de variable global.

Tabla 26 - RF-02

IDENTIFICADOR: RF-03
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta diferencia entra variables globales definidas con *constexpr*, con *const* o no constantes.
JUSTIFICACIÓN: Para poder eliminarlas primero hay que detectarlas.

Tabla 27 - RF-03

IDENTIFICADOR: RF-04
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta no modifica las variables globales definidas con constexpr.
JUSTIFICACIÓN: Estas constantes globales no suponen un problema puesto que el preprocesador se cargará de procesarlas en tiempo de compilación.

Tabla 28 - RF-04

IDENTIFICADOR: RF-05
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta no modifica las variables globales definidas entre macros.
JUSTIFICACIÓN: Las variables globales definidas entre macros no pueden eliminarse porque las macros no se van a eliminar y, por tanto, el comportamiento del programa se vería alterado.

Tabla 29 - RF-05

IDENTIFICADOR: RF-06
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta borra las variables globales que no están definidas como expresiones constantes (constexpr).
JUSTIFICACIÓN: Estas variables globales son las que se desea eliminar y por ello hay que borrarlas.

Tabla 30 - RF-06

IDENTIFICADOR: RF-07
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta reescribe las variables globales borradas según el RF-06 en el *main* del programa.
JUSTIFICACIÓN: Para que el programa funcione correctamente, las variables deben seguir existiendo, ahora de forma local en el main.

Tabla 31- RF-07

IDENTIFICADOR: RF-08
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta no modifica las variables globales externas (declaradas con extern).
JUSTIFICACIÓN: No suponen un problema.

Tabla 32- RF-08

IDENTIFICADOR: RF-09
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta añade los parámetros necesarios a las funciones que hacen uso de las variables globales borradas según el RF-06.
JUSTIFICACIÓN: Para que el programa funcione correctamente.

Tabla 33- RF-09

IDENTIFICADOR: RF-10
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta añade como parámetro de las llamadas a una función las variables globales borradas según el RF-06 que se utilicen en dicha función.
JUSTIFICACIÓN: Para que el programa funcione correctamente.

Tabla 34- RF-10

IDENTIFICADOR: RF-11
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta reescribe como expresión constante global (constexpr) aquellas variables globales que cumplan las características necesarias para ser constexpr. En su defecto, se aplicará RF-07.
JUSTIFICACIÓN: Para que el programa funcione correctamente.

Tabla 35- RF-11

IDENTIFICADOR: RF-12
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta reescribe como constante (const) una variable global que se va a eliminar de acuerdo con el RF-07 si no se modifica nunca su valor y si no es posible el RF-11.
JUSTIFICACIÓN: declarar como constante algo que es constante pero no estaba declarado como constante.

Tabla 36- RF-12

IDENTIFICADOR: RF-13
PRIORIDAD: Media
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta reescribe como expresiones constantes globales (constexpr) las constantes definidas mediante macros, respetando el RF-05.
JUSTIFICACIÓN: usar el estándar C++11.

Tabla 37- RF-13

IDENTIFICADOR: RF-14
PRIORIDAD: Media
NECESIDAD: Opcional
TIPO: Funcional
DESCRIPCIÓN: La herramienta informa al usuario del resultado de la ejecución a través del terminal.
JUSTIFICACIÓN: Siempre es conveniente informar al usuario del éxito o fracaso de una ejecución y del resultado.

Tabla 38- RF-14

IDENTIFICADOR: RF-15
PRIORIDAD: Media
NECESIDAD: Opcional
TIPO: Funcional
DESCRIPCIÓN: La herramienta no eliminará aquellas variables globales que tengan una redeclaración local en el *main*.
JUSTIFICACIÓN: El programa no funcionaría correctamente.

Tabla 39- RF-15

IDENTIFICADOR: RF-16
PRIORIDAD: Alta
NECESIDAD: Obligatoria
TIPO: Funcional
DESCRIPCIÓN: La herramienta no modificará ningún fichero fuente si detecta algún tipo de error o incompatibilidad.
JUSTIFICACIÓN: Se generaría código erróneo.

Tabla 40- RF-16

4. Diseño

En este apartado del documento se presenta el diseño de la herramienta siguiendo las necesidades descritas en la parte de análisis. Se presentará la arquitectura elegida y las decisiones de diseño tomadas antes de la implementación de la herramienta.

4.1 Arquitectura del sistema

Para definir la arquitectura del sistema se ha decidido dividir el sistema en diferentes capas que se comunican con las capas inmediatamente superiores e inferiores. La arquitectura se divide en 2 capas de la siguiente manera:

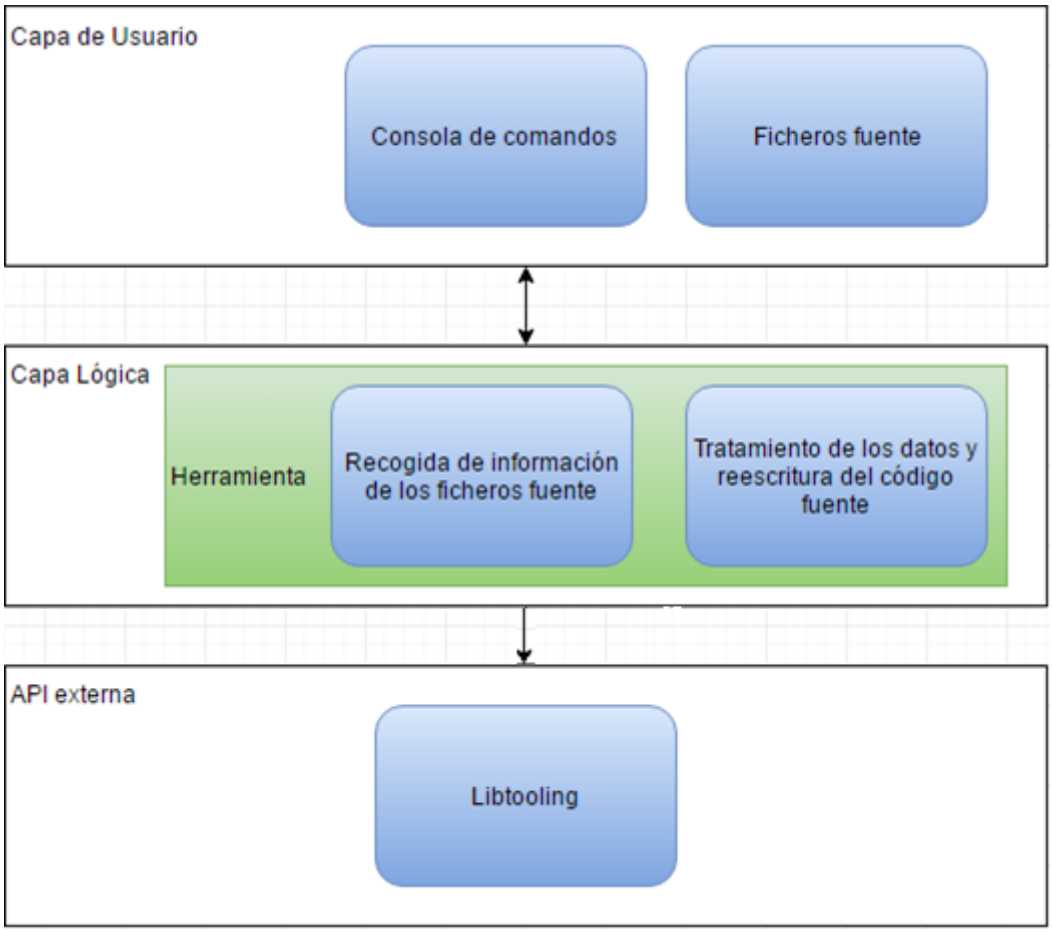

Ilustración 7 - Arquitectura

Como se puede observar en el diagrama anterior, la arquitectura cuenta con 2 capas horizontales que son la capa de usuario y la capa de lógica. Ambas capas tienen relación entre sí. La capa lógica, por su parte, hace uso de un API externo, el API de LibTooling.

Capa de usuario

La capa de usuario es la capa con la que el usuario interactúa. La herramienta carece de interfaces visuales programadas por lo que el usuario interactuará con la herramienta a través de la consola de comandos. A través de la consola de comandos el usuario ejecutará la herramienta y podrá pasarle por parámetros la ruta completa del fichero fuente escrito en C++ que desee.

Por otro lado, la capa lógica enviará a la capa de usuario información, es decir, la herramienta imprimirá por la consola de comandos los resultados de la ejecución. Por esta razón la relación entre ambas capas es bidireccional.

Capa Lógica

La capa de lógica es la que contiene la funcionalidad fundamental de la herramienta. Esta capa podemos separarla en dos módulos o tareas bien diferenciadas que se explicarán a continuación.

Uno de los módulos es el de recogida de información del código fuente. En esta tarea se extrae toda la información relevante del código fuente como variables globales, localizaciones, funciones en las que se usan las variables globales, etc. Toda la información relevante se almacena de forma temporal durante la ejecución de la herramienta. No hay persistencia de datos.

El otro módulo es más algorítmico. Analiza los datos obtenidos y realiza unas acciones u otras en función de la información que se está tratando. Finalmente reescribe el fichero de código fuente.

Para realizar estas tareas la capa de lógica hace uso de un API externa, el API de LibTooling proporcionado por Clang. Este API proporciona todas las funciones necesarias para extraer la información del código fuente. En este caso la relación entre la capa lógica y la API externa es unidireccional ya que solo la capa lógica hace uso de la API.

Como no existe la necesidad de persistencia de datos no hay capa de datos.

4.2 Especificación del entorno tecnológico

Los distintos componentes necesarios para el uso y el funcionamiento de la aplicación requieren de unos requisitos **mínimos** hardware y software. En este apartado definiremos estos requerimientos mínimos para poder utilizar y desarrollar la herramienta.

2.4.1 Hardware

Los requisitos hardware mínimos necesarios son los siguientes:

Cliente:

- El computador debe tener, al menos, 15 Gb libres en el disco duro.
- El computador debe tener, al menos, 4 Gb de memoria RAM.
- El computador debe tener un procesador Intel Core Duo T2400 o superior.

Entorno de desarrollo:

- 4 Gb RAM.
- Intel Core i3 4005U.
- 500 Gb de almacenamiento.
- Conexión wifi y Ethernet.
- Al menos un puerto USB 2.0.

2.4.2 Software

Los requerimientos mínimos software necesarios son los siguientes:

Cliente:

- Sistema operativo Ubuntu 14.04 LTS o superior.
- Clang 3.6 o superior.
- Clang tools.

Entorno de desarrollo:

- Sistema operativo Ubuntu 14.04 LTS o superior.
- Clang 3.6 o superior.
- Clang tools.
- Editor de texto Gedit.

4.3 Decisiones de diseño

En este apartado se explican y justifican algunas decisiones de diseño tomadas previa implementación de la herramienta, es decir, referentes a las tecnologías que se van a utilizar para desarrollar la herramienta.

Para implementar esta herramienta o cualquier otra herramienta que implique una modificación de los ficheros de código fuente no existen muchas alternativas posibles. Por un lado, tenemos Spoon [26], una librería de código abierto que analiza el código fuente y permite modificarlo. El problema, es para Java y no para C++ por lo que queda automáticamente descartada. Por otro lado, encontramos las distintas herramientas que ofrece Clang, estas sí son para C++ y para C. No hemos encontrado otra alternativa posible por lo que la herramienta se hará con las librerías que ofrece Clang.

Al seleccionar las librerías de Clang como opción de desarrollo estamos obligados a usar Clang como compilador. La herramienta a implementar ha de ser compilada con Clang ya que es el único compilador que entiende las librerías que nos van a permitir desarrollar la herramienta.

Clang ofrece varias interfaces para implementar herramientas de análisis y modificación del código fuente. Cada una de ellas tiene unas características que las hacen mejores para solventar unas necesidades u otras. Tenemos tres interfaces diferentes:

- **LibClang:** es una interfaz escrita en C con un alto nivel de abstracción. Recomendada cuando no se va a hacer uso de las características de C++ o cuando no se requiere de un control total del AST. Es la interfaz recomendada en caso de duda. [11]

- **Clang Plugins:** permite realizar acciones adicionales en el AST como parte del proceso de compilación. Los plugins son librerías dinámicas que se cargan en tiempo de ejecución por el compilador. Son dependientes del entorno para el que se crean por lo que no se recomiendan para herramientas independientes. [11]

- **LibTooling:** es una interfaz escrita en C++ cuyo objetivo es el de facilitar el desarrollo de herramientas independientes y su integración con los servicios de clang. Su uso se recomienda cuando se requiere de un control completo sobre el AST o cuando se quiere ejecutar la herramienta sobre un determinado fichero o conjunto de ficheros. [11]

Teniendo en cuenta que debemos respetar los requisitos formulados en la fase de análisis, la interfaz que mejor se adapta a nuestras necesidades es la de Libtooling ya que está escrita en C++ y nos va a permitir obtener toda la información del código escrito en C++. Además, permite crear las herramientas de forma independiente y permite un control completo sobre el AST, es decir, es menos restrictivo y nos ofrece más posibilidades, aunque será más complejo de manejar. LibClang queda descartada porque está en C y nosotros necesitamos C++ y el desarrollo en forma de plugin no encaja ya que queremos desarrollar una herramienta independiente.

5. Implementación

Una vez realizados el análisis y el diseño de la herramienta podemos pasar a la implementación de la misma. La implementación consiste en la programación de la herramienta eliminadora de variables globales. Para facilitar la explicación iremos explicando clase por clase de forma separada.

5.1 Instalación y configuración

Antes de empezar a desarrollar la herramienta es necesario obtener todo el software necesario para poder desarrollarla. Atendiendo a los requisitos definidos en el documento contamos con un computador con un hardware superior al requerido y con sistema operativo Ubuntu 15.10.

Partiendo de este punto lo primero es descargar el LLVM-Clang a través del terminal. Para ello vamos a descargar todos los archivos necesarios desde los repositorios de git, por tanto, es necesario tener instalado git mediante el comando **sudo apt-get install git**. Una vez obtenemos git podemos descargar los repositorios que queramos desde el terminal de Ubuntu, para ello escribimos en consola los siguientes comandos:

- **Descargar llvm**: git clone http://llvm.org/git/llvm.git
- **Movernos al directorio llvm/tools**: cd llvm/tools
- **Descargar clang**: git clone http://llvm.org/git/clang.git
- **Movernos al directorio llvm/projects**: cd ../projects
- **Descargar compiler-rt**: git clone http://llvm.org/git/compiler-rt.git
- **Movernos al directorio llvm/tools/clang/tools**: cd ../tools/clang/tools
- **Descargar clang tools extra**: git clone http://llvm.org/git/clang-tools-extra.git

Una vez realizados estos pasos ya hemos descargado todos los archivos necesarios. El siguiente paso es compilar todos estos archivos haciendo uso de cmake. Descargamos el cmake con el comando **sudo apt-get install cmake**.

No podemos compilar los archivos en el directorio principal de LLVM por lo que hay que crearse uno y hacerlo en ese directorio. Para ello:

- mkdir -p llvm-build
- cd llvm-build

Finalmente ejecutamos los siguientes comandos para compilar e instalar llvm y tener todo preparado para poder desarrollar la herramienta:

- cmake ../llvm -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX:PATH=<ruta de instalación>
- make -j <número de cores>
- make install

5.2 Implementación

Tras realizar el análisis y el diseño de la herramienta y tener todo el entorno preparado para el desarrollo podemos empezar el proceso de implementación. La implementación de la herramienta se realizará en un solo fichero escrito en el lenguaje de programación C++. La herramienta implementa diferentes clases y cada clase diferentes métodos. Para simplificar la explicación se explicará el desarrollo clase a clase.

Estructuras de datos globales del programa

- **bool valido:** variable global que determina si se puede o no llevar a cabo la reescritura del código. Si su valor es false, no se reescribe el código.
- **vector<SourceLocation> definiciondefineend:** Vector del tipo SourceLocation que sirve para almacenar la localización de la posición siguiente al último carácter de una macro. Sirve para borrar una macro.
- **vector<SourceLocation> ifsmacros:** Vector del tipo SourceLocation que sirve para almacenar la localización de macros del tipo *#ifdef* o *#ifndef*.
- **vector<SourceLocation> endifsmacros:** Vector del tipo SourceLocation que sirve para almacenar la localización de macros del tipo *#endif*.
- **vector<string> defineToVar:** Vector del tipo string que almacena la definición de una expresión constante que sustituye a una macro que define una constante.
- **vector<Token> vtok:** vector de tipo Token que sirve para almacenar tokens de expansiones de macros del tipo *#define.*
- **vector <MacroDefinition> vmd:** vector del tipo MacroDefinition que sirve para almacenar una definición de una macro del tipo *#define*.
- **vector<int> voffset:** Vector del tipo int que sirve para almacenar el tamaño o número de caracteres que conforma una macro. Sirve para borrar una macro.
- **string ruta:** String que almacena la ruta en la que se encuentra el fichero de código fuente pasado como parámetro en la ejecución**.**

5.2.1 Clase Find_Includes

Esta clase hereda de la clase PPCallbacks [27] de clang. Esta clase provee un conjunto de funciones que, al ser implementadas, se ejecutan bajo una cierta condición. La clase ofrece una forma de obtener información acerca de las acciones que realiza el preprocesador. Las funciones implementadas en esta clase nos permitirán localizar las macros de definición de constantes y localizar las fracciones de código que ocupa un par de macros *#ifdef/#endif* o *#ifndef/#endif*. De esta forma podemos resolver todo lo referente al código que trata el preprocesador. A continuación, se detallan las funciones implementadas:

Función MacroExpands (const Token &MacroNameTok, const MacroDefinition &MD, SourceRange Range, const MacroArgs *Args)

Esta función se ejecutará cuando se produzca una expansión de una macro, es decir, cuando se haga referencia a un *#define* existente. Dentro de ella vamos a guardar 2 cosas:

- La definición de la macro, en el vector global del programa *vmd* de tipo *MacroDefinition*.
- La expansión de la macro como un token[1], en el vector global del programa *vtok* de tipo *Token*.

Función Ifdef (SourceLocation Loc, const Token &MacroNameTok, const MacroDefinition &MD)

Esta función se ejecutará cuando el preprocesador detecte una macro del tipo *#ifdef.* Lo que nos interesa almacenar aquí es la localización de la macro, la cual almacenaremos en el vector global del programa *ifsmacros* de tipo *SourceLocation*.

Función Ifndef (SourceLocation Loc, const Token &MacroNameTok, const MacroDefinition &MD)

Esta función se ejecutará cuando el preprocesador detecte una macro del tipo *#ifndef.* Lo que nos interesa almacenar aquí es la localización de la macro, la cual almacenaremos en el vector global del programa *ifsmacros* de tipo *SourceLocation*.

Función Endif (SourceLocation Loc, SourceLocation IfLoc)

Esta función se ejecutará cuando el preprocesador detecte una macro del tipo *#endif.* Lo que nos interesa almacenar aquí es la localización de la macro, la cual almacenaremos en el vector global del programa end*ifsmacros* de tipo *SourceLocation*.

Después de *#ifdef* o *#ifndef* siempre va la macro *#endif,* que significa el cierre. Las localizaciones de las macros *#ifdef* y *#ifndef* se almacenan en el mismo vector, mezcladas. Esto no supone ningún problema ya que al tener el otro vector para las localizaciones de las macros *#endif* lo que se consigue es tener el par *#ifdef/#endif* o *#ifndef/#endif* en el mismo offset dentro del vector. Es decir, con el mismo índice en los distintos vectores accedemos al par de inicio y cierre correspondiente. Por tanto, los vectores tendrán una forma similar a la siguiente:

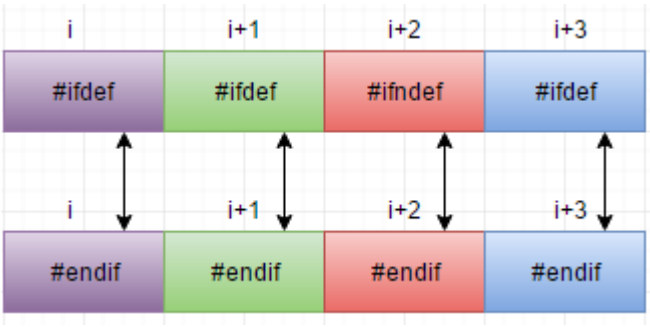

Ilustración 8 - Ejemplo almacenamiento temporal de macros

5.2.2 Clase MyASTVisitor

Esta clase hereda de la clase RecursiveASTVisitor [28] de clang. La clase RecursiveASTVisitor realiza un recorrido en profundidad de todos los nodos que componen el AST proporcionado por Clang, permitiéndonos sacar información relevante de cada nodo.

Esta clase implementa el grueso de la herramienta, se encarga de obtener la información referente al código fuente del usuario, clasifica la información y realiza la reescritura del código. Como se comentó en el diseño, la capa lógica se podía separar en 2 módulos uno de obtención de información del código fuente y otro de algorítmica y reescritura de datos. Siguiendo esa idea, se va a proceder a explicar primero la parte de extracción de información de los ficheros do código fuente.

Para la obtención de información, existen una serie de funciones en la clase RecursiveASTVisitor que se ejecutan cuando un nodo del AST posee una característica específica. Se implementa también una función propia dentro de la clase, la función *operaciones()* la cual se ejecutará con una llamada manual desde la clase

MyASTConsumer. A continuación, se describen cada uno de estos métodos utilizados para la obtención de información y las estructuras de datos de la clase.

Estructuras de datos globales de la clase

- **vector<VarDecl*> var_glob_cte:** vector que almacena las variables globales constantes.
- **vector<VarDecl*> var_glob_nocte:** vector que almacena las variables globales no constantes.
- **vector<string> lista_global_modif:** vector que almacena las variables modificadas a lo largo del programa.
- **vector<CallExpr *> lista_callExp:** almacena el conjunto de llamadas a funciones con parámetros.
- **vector<CallExpr *> all_callExpr**: vector que almacena todas las llamadas a funciones en el programa.
- **vector<FunctionDecl *> lista_func:** almacena todas las declaraciones de funciones que necesitan argumentos**.**
- **vector<FunctionDecl *> func:** almacena todas las funciones, servirá para reescritura de variable globales a locales y reescritura de funciones.
- **vector<vector<DeclRefExpr *>> lista_declrefexpr_functiondecl:** almacena vectores que contienen las variables que hay en cada función.
- **vector<vector<CallExpr *>> lista_callexpr_functiondecl:** almacena vectores de las callexpr que hay en cada función.
- **vector<vector<string>> tipo_declrefexpr:** almacena información de cada variable que hay en las funciones.
- **int cglob = 0:** contador de variables globales.
- **bool hay_globales:** variable que valdrá true en caso de haber alguna variable global en el programa.
- **SourceRange loc_main:** variable que almacena la posición en la que se localiza el *main* del programa.
- **vector<string> new_fundecls:** vector que almacena un string con la cadena a insertar en el código fuente para la reescritura de funciones.
- **vector<string> new_callexpr:** vector que almacena un string con la cadena a insertar en el código fuente para la reescritura de llamadas a funciones.

Antes de proceder con la explicación de cada una de las funciones es necesario explicar la estrategia seguida con los vectores que almacenan la información relativa a las funciones, variables y macros.

La idea es tener diferentes vectores que almacenan distinto tipo de información pero que a su vez están directamente relacionados. Los vectores que tengan relación entre sí, almacenaran en la misma posición del vector la información perteneciente a un elemento.

Para comprenderlo mejor se ha realizado un gráfico ilustrativo. En este gráfico tenemos 5 vectores que almacenan información relativa a las funciones. En la primera posición, se almacena toda la información referente a la función sumar(), en la segunda toda la referente a la función restar(), y así con tantas posiciones como funciones haya. De esta forma, si estamos iterando el vector *func*, mediante un contador podemos acceder a la misma posición del vector que se está iterando de forma directa en los demás vectores mediante el operador [i], y así, obtener toda la información referente a una función.

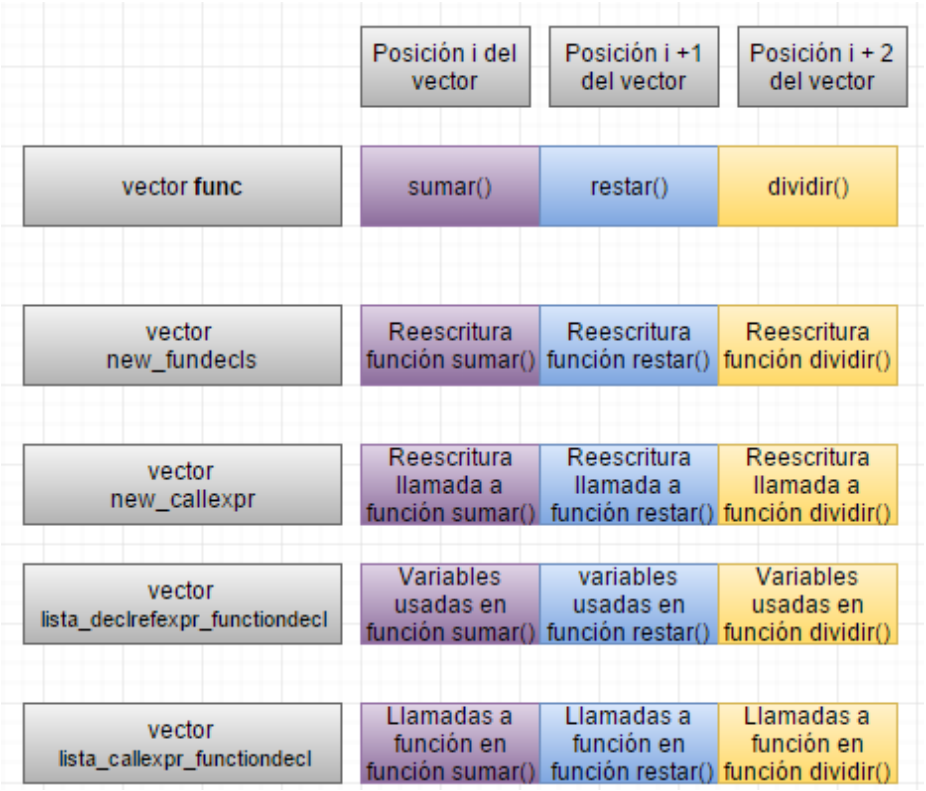

Ilustración 9 - Ejemplo almacenamiento temporal de los datos a tratar

De la misma forma se aplica esta forma de almacenamiento para la información de variables que hay que borrar y reescribir o de macros que hay que borrar y reescribir.

Función VisitDecl(Decl *D)

Este método se ejecutará cuando un nodo del AST sea una declaración o definición de cualquier tipo. Este método es el encargado de encontrar las variables globales del programa sobre el que actúa la herramienta, así como de clasificarlas.

Lo primero que realiza esta función es comprobar si la declaración está entre macros de preprocesador como puede ser #ifndef y #endif. En caso de estar declarada entre macros vamos a ignorar esta declaración ya que no vamos a modificar este tipo de variables globales puesto que habría que hacer modificaciones también sobre las macros, tarea bastante compleja y que no es objetivo de la herramienta. En caso de no estar declarada entre macros se prosigue con más comprobaciones.

El siguiente paso es realizar varias comprobaciones, que, de cumplirse, tenemos una variable global a tratar. Las comprobaciones son las siguientes:

- La declaración tiene ámbito global.
- La declaración no es externa (extern): si es externa la ignoramos, no se tratan.
- La declaración no es constexpr: no queremos modificar las variables globales definidas como expresiones constantes ya que estas no suponen un problema.

Si se dan estas condiciones estamos ante una variable global. Lo siguiente sería comprobar si está declarada como constante o no y almacenarlas en dos vectores diferentes uno que almacena las variables globales constantes y otro que almacena las variables globales no constantes. En caso de ya existir esta variable en los vectores de almacenamiento, no se vuelve a almacenar. Los vectores son de VarDecl, el tipo que representa la declaración de variables.

Función VisitStmt(Stmt *s)

El método VisitStmt se ejecutará cuando un nodo del AST sea un *statement,* normalmente expresiones. Este método se encarga de dos cosas, la primera de ellas consiste en detectar si una variable global se modifica o no a lo largo del programa. Esto nos servirá básicamente para poder declarar como constante una variable global que no se ha declarado como constante pero que no se modifica durante la ejecución del programa. Todas las variables modificadas serán almacenadas en un mismo vector, vector de strings que almacenará solamente el nombre de la variable. La segunda tarea que realiza es almacenar en vectores información referente a funciones y llamadas a funciones.

Entrando más en detalle, lo primero que realiza el método es comprobar si estamos ante una expresión simple (expr) ya que las expresiones indican modificaciones. En caso de ser una expresión se realizan múltiples comprobaciones referentes a los distintos tipos

posibles de modificación de una variable que se pueden dar. A continuación, se enumeran y detallan las distintas comprobaciones:

- **Si la expresión es un operador unitario**: esto quiere decir que estamos ante una modificación del tipo *++r* o *–r*. Para obtener la variable que está siendo objeto de esta modificación hay que obtener la sub expresión asociada a la expresión que está siendo tratada y de ahí obtener el nombre de la variable a la que está haciendo referencia. Para ello se hace uso del método *getSubExpr()*. Finalmente se almacena el nombre en el vector de variables modificadas.
- **Si la expresión es un operador binario:** en este caso estamos ante modificaciones del tipo *a = b* o *a = 3 + 4*. Para obtener la variable que se está medicando hay que obtener la expresión que está en el lado izquierdo del operador con la función *getLHS()*. Finalmente se almacena el nombre de la variable modificada en el vector de variables modificadas.
- **Si la expresión es del tipo *array subscript:*** este caso se da en modificaciones de arrays del tipo *arr[0]=1*. Para obtener la variable modificada se hace uso de la función *getBase()->IgnoreParenCasts()* que nos permite obtener la referencia a la declaración de la variable modificada y con ello a su nombre.
- **Si la expresión es del tipo *Memberexpr:*** estas expresiones hacen referencia a modificaciones en estructuras de datos(struct). Para obtener la variable modificada se hace uso de la función *getBase()->IgnoreParenCasts()* que nos permite obtener la referencia a la declaración de la variable modificada y con ello a su nombre.

En caso de no estar ante una expresión simple, existen otros tipos de expresiones que también debemos tratar y son aquellas referentes a objetos y contenedores. Para modificaciones de objetos y variables también hay expresiones específicas.

Si el *statement* es una expresión del tipo *CXXMemberCallExpr,* estamos ante una posible modificación de un objeto a través de una función propia. La detección de modificaciones de un objeto está bastante limitada y consiste en comprobar si el método del objeto al que se está llamando está declarado como constante o no. Si no está declarado como constante asumiremos que hay modificaciones en el objeto y almacenaremos el nombre del objeto modificado en el vector de variables modificadas.

El último caso de modificación de variables es la modificación de contenedores. Si el *statement* es una expresión del tipo *CXXOperatorCallExpr,* estamos ante una modificación de un contenedor a través de una función propia como pueda ser un push_back(). Obtendremos el nombre del contenedor con la siguiente función *getRawSubExprs()[1]* y lo almacenaremos en el vector de variables modificadas.

Como se comentó anteriormente, la segunda parte de este método se encarga de recoger la información referente funciones y llamadas a funciones. Concretamente, lo que hace es almacenar en un vector todas las llamadas a funciones que hay en el código y todas las funciones que tienen como parámetro un valor por referencia. El objetivo de esto es poder detectar la

modificación de variables globales que por alguna extraña razón se están pasando por referencia o por puntero en los parámetros de una función.

Para ello lo primero que se comprueba es si el *statement* es una expresión del tipo *CallExpr,* que son aquellas expresiones de llamadas a función. Si estamos ante una llamada a función nos interesa primero obtener la declaración de la función, mediante el método *getDirectCallee(),* para comprobar si tiene parámetros. Si no tiene parámetros no se realiza ninguna acción. Si tiene parámetros, hay que recorrerlos y preguntar si es de tipo puntero o referencia. En caso afirmativo, se almacena en un vector de FunctionDecl la declaración de la función.

Función VisitFunctionDecl(FunctionDecl *f)

Esta función se ejecutará cuando un nodo del AST sea una declaración o definición de una función. Esta función realiza dos tareas sobre aquellas funciones que sean funciones específicas del programa, es decir, funciones que no estén definidas dentro de clases, estructuras o contenedores.

La primera de ellas, cuando el nodo tratado es la definición del *main(),* se encarga de comparar si hay alguna variable local del *main* que tenga el mismo nombre que una variable global. Si se diera el caso, esta variable global se eliminaría del vector de variables globales al que pertenece y no será tratada. Esto se hace así porque si ya está definida localmente en el *main* no se puede volver a declarar una variable con el mismo nombre.

La segunda tarea se aplica al resto de funciones del programa y consiste en buscar todas las llamadas a funciones y variables que se utilizan en esa función. El objetivo de esto es recoger la información necesaria para, posteriormente, saber que variables globales se usan en una función. Primero comprobamos que el nodo tratado es una definición y no una declaración de una función comprobando si tiene cuerpo de la función. Si se cumple, se almacena la siguiente información en diferentes vectores globales de la clase:

- La declaración de la función en un vector de FunctionDecl.
- Un vector(**lista_declrefexpr_functiondecl)** que almacena todas las variables usadas dentro de la función en un vector de vectores de DeclRefExpr.
- Un vector(**lista_callexpr_functiondecl)** que almacena todas las llamadas a funciones realizadas dentro de la función en un vector de vectores de CallExpr.
- Un string vacío en el vector new_callexpr, para añadir los nuevos parámetros de una función.
- Un string vacío en el vector **new_fundecls**, para añadir los nuevos parámetros de una llamada a función.

Función TraverseDecl (Decl *D)

Esta función permite visitar de forma recursiva los nodos del AST a partir de una declaración. A esta función se le pasará el fichero entero como un Decl y se encargará de recorrer el AST completo.

Función operaciones()

Este método se encarga del tratamiento de los datos recogidos anteriormente y de reescribir el código de la forma que corresponda. Es una función muy extensa que realiza muchas operaciones diferentes, las cuales vamos a ir describiendo en orden de ocurrencia. Es necesario comentar que para todas las modificaciones del código fuente (borrado y escritura) se hace con la clase *Rewriter* proporcionada por Clang.

Detección de posibles modificaciones en variables globales pasadas por parámetro a funciones

Lo primero que realiza la función es detectar las variables globales que se pasan como referencia por parámetro a una función. Anteriormente, en la función VisitStmt, detectamos las modificaciones que se hacían de forma directa sobre una variable global, pero esto no incluye el caso que se está tratando ahora. Esto se consigue realizando los siguientes pasos:

1. Mediante un bucle anidado recorremos los vectores *lista_func* y *lista_callExpr* y comparamos en cada iteración si el nombre de la función coincide con el de la llamada a función. En caso de coincidir estamos ante un posible caso de lo que estamos buscando.
2. Si coinciden los nombres, recorrer mediante un bucle todos los parámetros de la llamada a la función.
3. Para cada parámetro, consultar si es un puntero o una referencia.
4. Si el parámetro es un puntero o una referencia a variable, se inserta en la lista de variables modificadas.

Notar que se trabaja con los parámetros de la llamada a función y no con los parámetros de la definición de la función, es decir, con las variables que se están pasando realmente como parámetro.

Recogida de información para la eliminación de constantes definidas mediante macros

La segunda tarea que realiza la función es reunir la información necesaria para sustituir macros del tipo #define test 3 a expresiones constantes del tipo *constexpr auto test = 3*. El tratamiento de macros en clang es bastante complejo y hay que tener en cuenta varias cosas a la hora de implementar cosas sobres las macros. Una de ellas es el control sobre lo que queremos modificar. Esta herramienta tiene como objetivo modificar solamente los ficheros fuente del usuario, pero el preprocesador de clang trabaja sobre todos los ficheros que se hacen uso en el programa. Si el programa hace uso de *cmath*, las

funciones de la clase *PPCallbacks* también se van a ejecutar para las macros definidas en los ficheros de *cmath*.

Para solventar esto y abarcar solo los ficheros que nos competen hay que obtener la ruta del fichero en la que se encuentra definida la macro. Esto se debe obtener de cada MacroDefinition del vector global *vmd.* En caso de que esa macro este definida en un fichero cuyo directorio es el mismo que el introducido como parámetro en la ejecución del programa estamos ante una macro a modificar.

Una vez sabemos si tenemos que tratar o no la macro, en caso afirmativo, hay que obtener los datos de los que se compone la macro. Esto se puede obtener a partir del *Token* almacenado en el vector *vtok*. Obtenido el nombre y valor del #define, hay que comprobar que estamos ante una definición de una constante (#define test 3) y no ante un renombramiento de un tipo de dato (#define ftype float).

Si se estuviera renombrando un dato, estamos ante un caso excepcional en el que la herramienta no puede reescribir el código ya que es incompatible con este tipo de macros. Por ello, hay que poner el valor *false* en la variable global *valido* para evitar realizar la reescritura.

En caso de estar ante una definición de una constante, hay que realizar 4 acciones:

- Crear el string con la definición de la expresión constante global que sustituirá a la macro. El string tendrá la siguiente forma:
  "constexpr auto "+nombre_del_define+ " = "+ valor_del_define+";".
- Almacenar este string en el vector global *defineToVar.*
- Almacenar en el vector global *definiciondefineend* la localización de la posición en la que acaba la macro. Esto servirá para poder borrar la macro más adelante.
- Almacenar en el vector *voffset* la longitud, en número de caracteres, que ocupa la macro. Esto servirá para poder borrar la macro más adelante.

Borrado y reescritura de variables globales no constantes

Esta parte es una de las más extensas y complejas ya que hay que tener en cuenta diversos factores. Esta parte se realiza para cada variable global no constante almacenada en el vector global de variables globales no constantes(VarDecl). Se irán estableciendo una serie de títulos a las distintas tareas que se realizan dentro de esta extensa parte para poder hacer referencia a las mismas de unos puntos a otros.

Obtención de características

Lo primero que se realiza es comprobar si la variable global era estática o no para no perder esa característica de la variable. En caso de ser estática guardamos en un string el valor *"static".* Lo segundo a realizar es comprobar si la variable global puede ser declarada como *constexpr,* en caso afirmativo se guardará otro string el valor *"constexpr".* Lo tercero y último a realizar antes de comenzar la reescritura es comprobar si la variable global se ha modificado o no desde su inicialización, es decir si puede ser constante o no.

Lo siguiente a realizar es diferenciar entre variables globales no constantes inicializadas y las no inicializadas y dentro de esta diferenciación, diferenciar también entre las que modifican su valor durante la ejecución o no. ya que la forma de modificarlas será diferente. Las variables globales inicializadas pueden tomar la característica de constante o expresión constante en caso de no ser modificadas, las no inicializadas no. No obstante, las diferencias vienen en lo que se va a reescribir, pero no en el algoritmo a seguir.

Borrado y reescritura

Para las variables globales no constantes inicializadas que no se modifican durante la ejecución se realiza lo siguiente:

1. Crear un objeto Rewriter de clang que permite la reescritura del código fuente[29].
2. Borrar la línea en la que está definida la variable global, siempre y cuando no se haya borrado antes. Para ello hay que calcular la longitud de la línea y la posición de inicio de la línea. Todo esto se obtiene del objeto VarDecl que se está tratando en la iteración.
3. Obtener el tipo de la variable mediante la función getType().
4. Obtener el valor de inicialización de la variable mediante la función getInit().
5. Obtener qué tipo de variable es, un objeto, una variable de tipo primitivo, un array u otro diferente. En función del tipo de variable se reescribirá la variable de una forma u otra. Para todas las variables que no vayan a reescribirse como una expresión constante global, se almacenará un string con la declaración, para la reescritura de los parámetros de una función, y otro string con el nombre de la variable, para pasarlos por parámetro en las llamadas a función. Para cada tipo de variable se realizará lo siguiente:
    a. Si es un puntero: además de guardar la declaración y el nombre de la variable, se reescribe el puntero en el comienzo de la función main el programa. No se declaran como constantes. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable = valor_inicialización;

b. Si es un array: además de guardar la declaración y el nombre de la variable, se reescribe el array en el comienzo de la función main del programa. No se declaran como constantes o expresión constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType()en dos partes usando como delimitador un espacio(“ “) y reordenarlo. El string a insertar en el código se compone de estática_o_no primera_parte_getType nombre_variable segunda_parte_getType = valor_inicialización;
c. Si es una estructura de datos: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe la estructura en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType() usando como delimitador un espacio(“ “) y únicamente hacer uso de la segunda parte del string. El string a insertar en el código se compone de estática_o_no const_o_constexpr segunda_parte_getType nombre_variable = valor_inicialización;
d. Si es un objeto de datos: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe el objeto en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType() usando como delimitador un espacio(“ “) y únicamente hacer uso de la segunda parte del string. El string a insertar en el código se compone de estática_o_no const_o_constexpr segunda_parte_getType nombre_variable = valor_inicialización;

e. Si es una variable de tipo primitivo: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe la variable en el comienzo de la función main del programa como constante. El string a insertar en el código se compone de estática_o_no const_o_constexpr tipo_dato nombre_variable = valor_inicialización;

Para las variables globales no constantes y no inicializadas que no se modifican durante la ejecución se realiza lo siguiente:

1. Crear un objeto Rewriter de clang que permite la reescritura del código fuente[29].
2. Borrar la línea en la que está definida la variable global, siempre y cuando no se haya borrado antes. Para ello hay que calcular la longitud de la línea y la posición de inicio de la línea. Todo esto se obtiene del objeto VarDecl que se está tratando en la iteración.
3. Obtener el tipo de la variable mediante la función getType().
4. Obtener qué tipo de variable es, un objeto, una variable de tipo primitivo, un array u otro diferente. En función del tipo de variable se reescribirá la variable de una forma u otra. Para todas las variables se almacenará un string con la declaración, para la reescritura de los parámetros de una función, y otro string con el nombre de la variable, para pasarlos por parámetro en las llamadas a función. Para cada tipo de variable se realizará lo siguiente:
    a. Si es un puntero: además de guardar la declaración y el nombre de la variable, se reescribe el puntero en el comienzo de la función main el programa. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable;
    b. Si es un array: además de guardar la declaración y el nombre de la variable, se reescribe el array en el comienzo de la función main del programa. Para reescribir el tipo correctamente hay que separar el string obtenido con getType()en dos partes usando como delimitador un espacio(“ “) y reordenarlo. El string a insertar en el código se compone de estática_o_no primera_parte_getType nombre_variable segunda_parte_getType ;
    c. En cualquier otro caso: además de guardar la declaración y el nombre de la variable, se reescribe la variable en el comienzo de la función main del programa. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable;

Notar que en esta última clasificación de variables globales no constantes y no inicializadas que no se modifican durante la ejecución no se reescriben como constantes porque no están inicializadas.

Para las variables globales no constantes inicializadas que se modifican durante la ejecución se realiza lo siguiente:

1. Crear un objeto Rewriter de clang que permite la reescritura del código fuente[29].

2. Borrar la línea en la que está definida la variable global, siempre y cuando no se haya borrado antes. Para ello hay que calcular la longitud de la línea y la posición de inicio de la línea. Todo esto se obtiene del objeto VarDecl que se está tratando en la iteración.

3. Obtener el tipo de la variable mediante la función getType().

4. Obtener el valor de inicialización de la variable mediante la función getInit().

5. Obtener qué tipo de variable es, un objeto, una variable de tipo primitivo, un array u otro diferente. En función del tipo de variable se reescribirá la variable de una forma u otra. Para todas las variables que no vayan a reescribirse como una expresión constante global, se almacenará un string con la declaración, para la reescritura de los parámetros de una función, y otro string con el nombre de la variable, para pasarlos por parámetro en las llamadas a función. Para cada tipo de variable se realizará lo siguiente:

   a. Si es un puntero: además de guardar la declaración y el nombre de la variable, se reescribe el puntero en el comienzo de la función main el programa. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable = valor_inicialización;
   b. Si es un array: además de guardar la declaración y el nombre de la variable, se reescribe el array en el comienzo de la función main del programa. Para reescribir el tipo correctamente hay que separar el string obtenido con getType() en dos partes usando como delimitador un espacio(“ “) y reordenarlo. El string a insertar en el código se compone de estática_o_no primera_parte_getType nombre_variable segunda_parte_getType = valor_inicialización;
   c. Si es una estructura de datos: además de guardar la declaración y el nombre de la variable, se reescribe la estructura en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType() usando como delimitador un espacio(“ “) y únicamente hacer uso de la segunda parte del string. El string a insertar en el código se compone de estática_o_no segunda_parte_getType nombre_variable = valor_inicialización;
   d. Si es un objeto: además de guardar la declaración y el nombre de la variable, se reescribe el objeto en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay

que separar el string obtenido con getType() usando como delimitador un espacio(“ “) y únicamente hacer uso de la segunda parte del string. El string a insertar en el código se compone de estática_o_no segunda_parte_getType nombre_variable = valor_inicialización;

e. Si es una variable de tipo primitivo: además de guardar la declaración y el nombre de la variable, se reescribe la variable en el comienzo de la función main del programa como constante. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable = valor_inicialización;

Para las variables globales no constantes y no inicializadas que se modifican durante la ejecución se realiza lo siguiente:

1. Crear un objeto Rewriter de clang que permite la reescritura del código fuente[29].

2. Borrar la línea en la que está definida la variable global, siempre y cuando no se haya borrado antes. Para ello hay que calcular la longitud de la línea y la posición de inicio de la línea. Todo esto se obtiene del objeto VarDecl que se está tratando en la iteración.

3. Obtener el tipo de la variable mediante la función getType().

4. Obtener qué tipo de variable es, un objeto, una variable de tipo primitivo, un array u otro diferente. En función del tipo de variable se reescribirá la variable de una forma u otra. Para todas las variables se almacenará un string con la declaración, para la reescritura de los parámetros de una función, y otro string con el nombre de la variable, para pasarlos por parámetro en las llamadas a función. Para cada tipo de variable se realizará lo siguiente:

a. Si es un puntero: además de guardar la declaración y el nombre de la variable, se reescribe el puntero en el comienzo de la función main el programa. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable;

b. Si es un array: además de guardar la declaración y el nombre de la variable, se reescribe el array en el comienzo de la función main del programa. Para reescribir el tipo correctamente hay que separar el string obtenido con

getType()en dos partes usando como delimitador un espacio(“ “) y reordenarlo. El string a insertar en el código se compone de estática_o_no primera_parte_getType nombre_variable segunda_parte_getType ;

c. En cualquier otro caso: además de guardar la declaración y el nombre de la variable, se reescribe la variable en el comienzo de la función main del programa. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable;

Una vez finalizado esta parte, tenemos la variable global borrada de su ámbito global y reescrita de forma local en la función *main* del programa o reescrita como expresión constante global. Además, tenemos los strings con la declaración y el nombre de la variable para la reescritura de funciones y llamadas a función. Este será el siguiente paso, introducir la variable global tratada en cada iteración como parámetro en aquellas funciones en las que se hace uso de la misma.

Generación de los datos de reescritura de funciones y llamadas a función

Para cada variable global que se está tratando se realizará este proceso. Para realizar este proceso, se iterará sobre el vector de funciones rellenado por la función **VisitFunctionDecl** anteriormente descrita**.** Se hará uso de los vectores new_fundecls y new_callexpr, en los que cada posición del vector contiene lo que hay que reescribir en la definición de una función y lo que hay que reescribir en las llamadas a esa función respectivamente. Es importante mencionar que la misma posición en todos los vectores que tienen relación pertenecen a la misma función. Por ejemplo, si la función sumar() es el primer elemento del vector en el que se almacenan las funciones, new_fundecls[0], new_callexpr [0], lista_declrefexpr_functiondecl[0] y lista_callexpr_functiondec[0] almacenan la información relativa a sumar(). De esta forma se puede acceder a la información del mismo elemento usando el mismo índice. Para cada función se realiza lo siguiente:

1. Para aquellas funciones que aún no han sido tratadas, es decir el valor de *new_fundecl[i]* es un string vacío, y además no tienen parámetros, es necesario reescribir la función entera. Por ello insertamos en la posición correspondiente a la función tratada un string compuesto por “tipo_return nombre_función (“. En caso de que la función tenga parámetros no es necesario hacer esto ya que los parámetros nuevos se insertarán después del último parámetro existente.
2. A continuación, hay que recorrer el vector de variables que se utilizan en esa función(lista_declrefexpr_functiondecl) y que se rellenó en la función VisitFunctionDecl.
3. Si la variable global no constante almacenada en el vector global de variables globales no constantes que se está tratando en el bucle principal es la misma que la de una iteración en este bucle interno, quiere decir que la variable global sobre

la que se está realizando todo el proceso se usa dentro de la función que se está tratando en la iteración del bucle de funciones.

4. Si estamos ante una variable global que se usa dentro de la función que se está tratando, hay que ampliar los strings que almacenan la información a reescribir en funciones y llamadas a funciones. En este punto hay que distinguir entre diferentes situaciones:
    a. Si la función original no tiene parámetros y aún no se ha escrito ningún parámetro, tendremos, para la cadena de la definición de la función(new_fundecls), la cadena que escribimos al principio: "tipo_return nombre_función (". Por ello, a esta cadena existente le concatenamos el string que almacena la declaración y que se rellenó durante del proceso de borrado y reescritura de variables globales descrito anteriormente. Se rellenará también la cadena de la llamada a esta función, que hasta este punto estaba vacía, con el nombre de la variable.
    b. Si la función original no tiene parámetros, pero ya se ha pasado por la situación del punto "a", le concatenamos una coma y el string que almacena la declaración al string existente en new_fundecls. Le concatenamos al string de new_callexpr una coma y el nombre de la variable.
    c. Si la función original tiene parámetros, pero no hemos añadido ningún parámetro nuevo, se copia en el string del vector new_funcdecls una coma y la declaración de la variable. También se copia en el string del vector new_callexpr una coma y el nombre de la variable.
    d. Si la función original tiene parámetros y se ha pasado por el punto "c", se concatena al string del vector new_funcdecls una coma y la declaración de la variable. También se concatena al string del vector new_callexpr una coma y el nombre de la variable.
5. Después de realizar el punto 4, lo siguiente a realizar es el tratamiento de un caso especial y muy concreto, cuando se realiza una llamada a una función que utiliza una variable global como parámetro dentro de una función. Es un caso especial ya que en el vector lista_declrefexpr_functiondecl, que almacena las variables usadas en una función, no se almacenan las variables utilizadas como parámetro de llamada a una función. Por esta razón, si una variable global solo se utiliza en una función para pasarla como parámetro en la llamada a otra función, no se realizaría el proceso de reescritura de parámetros de las funciones correctamente y el código quedaría erróneo. Si se produce esto, hay que reescribir la definición y las llamadas a función de las dos funciones que intervienen.

    Para resolver este caso, hay que recorrer el vector que almacena las llamadas a función que realiza cada función, que se almacenan en lista_callexpr_functiondecl. Si la función que se está tratando es llamada por alguna función se va a realizar el proceso descrito en el punto 4 para esa función

con la misma variable global que se está tratando. Finalmente, para evitar que esta función, que ha tratado esta variable global antes de que le tocara en su iteración correspondiente, vuelva a tratarla, borramos de la lista de variables globales utilizadas en esta función la variable global que acaba de tratar anticipadamente.

Finalizada esta última parte, acaba todo el proceso que se realiza sobre cada variable global no constante. Recapitulando, se ha realizado lo siguiente sobre cada variable global de este tipo:

- Borrado de la variable global
- Reescritura en el main si no es una expresión constante.
- La variable se añade como parámetro en la definición de todas aquellas funciones que la usan.
- La variable se añade como parámetro en las llamadas a función de todas aquellas funciones que la usan.

Borrado y reescritura de variables globales constantes

Esta parte es similar a la descrita anteriormente para las variables globales no constantes, pero menos extensa ya que solo es un caso a tratar. Esta parte se realiza para cada variable global constante almacenada en el vector global de variables globales constantes(VarDecl). A continuación, se describen las distintas tareas que se realizan.

Obtención de características

Lo primero que se realiza es comprobar si la variable global era estática o no para no perder esa característica de la variable. En caso de ser estática guardamos en un string el valor *"static".* Lo segundo a realizar es comprobar si la variable global puede ser declarada como *constexpr,* en caso afirmativo se guardará otro string el valor *"constexpr".*

Borrado y reescritura

Para las variables globales constantes se realizan los siguientes pasos:

1. Crear un objeto Rewriter de clang que permite la reescritura del código fuente[29].
2. Borrar la línea en la que está definida la variable global, siempre y cuando no se haya borrado antes. Para ello hay que calcular la longitud de la línea y la posición de inicio de la línea. Todo esto se obtiene del objeto VarDecl que se está tratando en la iteración.
3. Obtener el tipo de la variable mediante la función getType().

4. Obtener el valor de inicialización de la variable mediante la función getInit().
5. Obtener qué tipo de variable es, un objeto, una variable de tipo primitivo, un array u otro diferente. En función del tipo de variable se reescribirá la variable de una forma u otra. Para todas las variables que no vayan a reescribirse como una expresión constante global, se almacenará un string con la declaración, para la reescritura de los parámetros de una función, y otro string con el nombre de la variable, para pasarlos por parámetro en las llamadas a función. Para cada tipo de variable se realizará lo siguiente:
    a. Si es un puntero: además de guardar la declaración y el nombre de la variable, se reescribe el puntero en el comienzo de la función main el programa. El string a insertar en el código se compone de estática_o_no tipo_dato nombre_variable = valor_inicialización;
    b. Si es un array: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe el array en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType()en dos partes usando como delimitador un espacio(" ") y reordenarlo. El string a insertar en el código se compone de estática_o_no const_o_constexpr primera_parte_getType nombre_variable segunda_parte_getType = valor_inicialización;
    c. Si es una estructura de datos: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe la estructura en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType() usando como delimitador un espacio(" ") y únicamente hacer uso de la segunda parte del string. El string a insertar en el código se compone de estática_o_no const_o_constexpr segunda_parte_getType nombre_variable = valor_inicialización;
    d. Si es un objeto: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe el objeto en el comienzo de la función main del programa como constante. Para reescribir el tipo correctamente hay que separar el string obtenido con getType() usando como delimitador un espacio(" ") y únicamente hacer uso de la segunda parte del string. El string a insertar en el código se compone de estática_o_no const_o_constexpr segunda_parte_getType nombre_variable = valor_inicialización;

e. Si es una variable de tipo primitivo: si se va a reescribir como expresión constante, se reescribe como variable global. Si no se reescribe como expresión constante, además de guardar la declaración y el nombre de la variable, se reescribe la variable en el comienzo de la función main del programa como constante. El string a insertar en el código se compone de estática_o_no const_o_constexpr tipo_dato nombre_variable = valor_inicialización;

Finalizada esta parte, tenemos la variable global borrada de su ámbito global y reescrita de forma local en la función *main* del programa o reescrita como expresión constante global. Además, tenemos los strings con la declaración y el nombre de la variable para la reescritura de funciones y llamadas a función. Este será, de nuevo, el siguiente paso, introducir la variable global tratada en cada iteración como parámetro en aquellas funciones en las que se hace uso de la misma.

Generación de los datos de reescritura de funciones y llamadas a función

Este proceso es exactamente igual al descrito para las variables globales no constantes.

En este punto ya se han tratado todas las variables globales y se ha obtenido la información necesaria para reescribir las funciones y las llamadas a funciones que hacen uso de estas variables globales. La siguiente tarea dela función es la reescritura de las funciones y sus llamadas.

Reescritura de funciones

A continuación, se explica la lógica seguida para implementar la reescritura de funciones realizando lo descrito sobre cada función.

Lo primero es comprobar si la función original era estática o no para no perder esa característica. En caso de ser estática guardamos en un string el valor *"static".* De la misma forma, hay que comprobar si la función original estaba definida con la palabra clave *inline.* En caso afirmativo, guardamos en un string el valor *"inline"*.

Realizado esto ya tenemos toda la información necesaria para la reescritura. Se distinguen dos casos a la hora de reescribir las funciones. El primero de ellos se da cuando la función original no tiene parámetros. Al no tener parámetros la API no nos facilita la introducción de parámetros de una forma sencilla. Por eso se opta por borrar el fragmento de código que va desde que empieza la definición de la función hasta la llave que indica el inicio del cuerpo de la función. Una vez borrado este fragmento, insertamos antes de la llave del cuerpo de la función la nueva definición elaborada anteriormente y que se encuentra almacenada en el vector new_fundecls.

El segundo caso se produce cuando la función original si tiene parámetros. En este caso no es necesario borrar nada ya que los nuevos parámetros se escriben justo después del último parámetro existente. Por ello simplemente hay que recuperar la localización de

la última posición del último parámetro y escribir los nuevos parámetros de la función que se encuentran almacenados en el vector new_fundecls.

Reescritura de llamadas a función

A continuación, se explica la lógica seguida para implementar la reescritura de las llamadas a funciones, realizando lo descrito sobre cada llamada a función existente. El proceso es similar a la reescritura de funciones.

Al igual que en la reescritura de funciones, se distinguen dos casos a la hora de reescribir las llamadas a funciones. El primero de ellos se da cuando la función original no tiene parámetros. Al no tener parámetros la API no nos facilita la introducción de parámetros de una forma sencilla. Por eso se opta por borrar el fragmento de código que contiene la llamada a función. Una vez borrado este fragmento, insertamos la llamada a la función con sus nuevos parámetros en la misma posición en la que borramos el fragmento de la misma llamada. Esta nueva llamada a función se encuentra almacenada en el vector new_callexpr.

El segundo caso se produce cuando la función original si tiene parámetros. En este caso no es necesario borrar nada ya que los nuevos parámetros se escriben justo después del último parámetro existente. Por ello simplemente hay que recuperar la localización de la última posición del último parámetro de la llamada a la función y escribir los nuevos parámetros de la función en la siguiente posición a la recuperada. Los nuevos parámetros a escribir se encuentran almacenados en el vector new_callexpr.

Reescritura de constantes definidas mediante macros

En este último paso de modificación del código fuente, se realiza la transformación de las constantes globales definidas mediante macros a expresiones constantes globales. Para conseguir esto hay que hacer uso de la posición final de cada macro en el código y de la longitud de la misma en caracteres. Esta información se encuentra almacenada en los vectores *definiciondefineend y voffset*. Con estos dos elementos se calcula el fragmento de código fuente a borrar y la primera posición sobre la que escribir la nueva expresión constante. Una vez borrado el fragmento correspondiente de la macro, se inserta la nueva variable global como expresión constante en la primera posición de la macro borrada. El string que contiene la definición de la nueva variable se almacena en el vector *defineToVar*.

Confirmación de los cambios

Una vez realizadas todas las modificaciones, hay que realizar el volcado sobre el fichero. Para ello hay que hacer uso de la función *overwriteChangedFiles()* de la clase *Rewriter*. Si no se han detectado errores durante todo el procesado de datos y modificación de los mismos, se realizará la llamada a esta función, en caso contrario no se realizará y no se efectuarán cambios sobre los ficheros.

5.2.3 Clase MyASTConsumer

Esta clase hereda de la clase ASTConsumer [30] la cual permite leer ASTs. Esta clase se encarga de inicializar un objeto MyASTVisitor descrito anteriormente para recorrer los nodos del AST.

Únicamente implementa la función ***HandleTranslationUnit(ASTContext &Context)***, que se llama cuando se ha parseado todo el AST. A partir del contexto pasado por parámetro se puede obtener un objeto del tipo *TranslationUnitDecl* el cual representa el fichero fuente a tratar en forma de *Decl*. Este objeto *TranslationUnitDecl* se le pasará por parámetro a la función *TraverseDecl* del objeto *MyASTVisitor,* consiguiendo de esta forma recorrer todo el AST. Finalmente, la función llama a la función *operaciones()* del objeto *MyASTVisitor.*

5.2.4 Clase MyFrontendAction

Esta clase hereda de la clase ASTFrontendAction [31] la cual permite trabajar con el AST de un fichero de código fuente. Para cada fichero crea un objeto MyASTConsumer.

Además, implementa 2 funciones, una que se ejecutará antes de procesar el fichero y otra al finalizar.

Función BeginSourceFileAction(CompilerInstance &ci, StringRef)

Esta función se ejecutará antes de procesar un fichero de código fuente. La tarea de esta función es la de poner en funcionamiento la clase *Find_Includes(),* para obtener información relativa a las operaciones del preprocesador. Para ello hay que recuperar una instancia del preprocesador, crear un puntero a un objeto *Find_Includes()* y pasarle a la función *addPPCallbacks()* del preprocesador el puntero creado.

Función EndSourceFileAction()

Esta función se ejecuta después de procesar el fichero de código fuente. Se utiliza, fundamentalmente, para mostrar información. Por ejemplo, si se produjo un error durante la ejecución.

6. Pruebas y evaluación

Una vez finalizada la implementación de la herramienta es imprescindible verificar el correcto funcionamiento de la misma y ver que satisface las necesidades del cliente. Para asegurarse de esto es necesario realizar un conjunto de pruebas suficientes como para afirmar que la herramienta funciona correctamente. Para conseguir esto haremos uso de pruebas unitarias y de pruebas de integración.

6.1 Pruebas unitarias

Las pruebas unitarias son aquellas que permiten probar el correcto funcionamiento de una parte específica del software. Para describir cada una de las pruebas realizadas se utilizará una tabla como la siguiente:

Identificador: PR - XX	
Nombre:	Nombre de la prueba
Descripción:	Descripción de la prueba
Resultado esperado:	Resultado esperado
Resultado obtenido:	Satisfactorio o no satisfactorio.

Tabla 41 - Ejemplo tabla pruebas unitarias

A continuación, se muestran todas las pruebas realizadas:

Identificador: PR - 01	
Nombre:	Detección de variables globales
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global no constante • Una variable global constante (const). • Una variable global como expresión constante (constexpr). Imprimir por consola las trazas correspondientes.
Resultado esperado:	La herramienta detecta los diferentes tipos de variable globales.
Resultado obtenido:	Satisfactorio.

Tabla 42 - PR-01

Identificador: PR - 02	
Nombre:	Clasificación de variables globales
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global no constante • Una variable global constante (const). • Una variable global como expresión constante (constexpr). Imprimir por consola las trazas correspondientes.
Resultado esperado:	La variable global no constante se introduce en el vector de variables globales no constantes, la variable global declarada como const se introduce en el vector de variables globales constantes y la variable global declarada como constexpr no se almacena.
Resultado obtenido:	Satisfactorio.

Tabla 43 - PR-02

Identificador: PR - 03	
Nombre:	Detección de modificaciones mediante operadores unitarios
Descripción:	Generar un programa escrito en C++ que tenga una variable global inicializada. Su valor se modificará en el *main* del programa mediante el operador ++. Imprimir por consola las trazas correspondientes.
Resultado esperado:	La variable global ha sido modificada y se ha detectado la modificación mediante un operador unitario.
Resultado obtenido:	Satisfactorio.

Tabla 44 - PR-03

Identificador: PR - 04	
Nombre:	Detección de modificaciones mediante operadores binarios
Descripción:	Generar un programa escrito en C++ que tenga una variable global inicializada. Su valor se modificará en el *main* del programa mediante el operador de igualdad. Imprimir por consola las trazas correspondientes.
Resultado esperado:	La variable global ha sido modificada y se ha detectado la modificación mediante un operador binario.
Resultado obtenido:	Satisfactorio.

Tabla 45 - PR-04

Identificador: PR - 05	
Nombre:	Detección de modificaciones de arrays
Descripción:	Generar un programa escrito en C++ que tenga una variable global inicializada de tipo array. Su valor se modificará en el *main* del programa. Imprimir por consola las trazas correspondientes.
Resultado esperado:	La variable global ha sido modificada y se ha detectado la modificación.
Resultado obtenido:	Satisfactorio.

Tabla 46 - PR-05

Identificador: PR - 06	
Nombre:	Detección de modificaciones de estructuras de datos
Descripción:	Generar un programa escrito en C++ que tenga una estructura de datos y una variable global inicializada de esa estructura de datos creada. Su valor se modificará en el *main* del programa. Imprimir por consola las trazas correspondientes.
Resultado esperado:	La estructura de datos global ha sido modificada y se ha detectado la modificación.
Resultado obtenido:	Satisfactorio.

Tabla 47 - PR-06

Identificador: PR - 07	
Nombre:	Detección de modificaciones de objetos
Descripción:	Generar un programa escrito en C++ que tenga una clase definida y una variable global inicializada de ese objeto creado. Su valor se modificará en el *main* del programa. Imprimir por consola las trazas correspondientes.
Resultado esperado:	La variable global ha sido modificada y se ha detectado la modificación.
Resultado obtenido:	Satisfactorio.

Tabla 48 - PR-07

Identificador: PR - 08	
Nombre:	Detección de modificaciones de contenedores
Descripción:	Generar un programa escrito en C++ que tenga vector de ámbito global inicializado con algún valor. Posteriormente, se introducen en el vector más valores, en el *main* del programa. Imprimir por consola las trazas correspondientes.
Resultado esperado:	El vector de ámbito global ha sido modificad y se ha detectado la modificación.
Resultado obtenido:	Satisfactorio.

Tabla 49 - PR-08

Identificador: PR - 09	
Nombre:	Detección de modificaciones por parámetros
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global inicializada de tipo *int.* • Una función que reciba como parámetro una referencia a un *int*. Posteriormente, en el main del programa, llamar a la función pasando como parámetro la variable global creada. Imprimir por consola las trazas correspondientes.
Resultado esperado:	Se ha detectado la posible modificación al pasar una variable global como parámetro, por referencia, a una función.
Resultado obtenido:	Satisfactorio.

Tabla 50 - PR-09

Identificador: PR - 10	
Nombre:	Eliminación de variables globales no constantes
Descripción:	Generar un programa escrito en C++ que tenga una variable global no constante cuyo valor es modificado en el main.
Resultado esperado:	La variable global se ha borrado.
Resultado obtenido:	Satisfactorio.

Tabla 51 - PR-10

Identificador: PR - 11	
Nombre:	Eliminación de variables globales constantes(const)
Descripción:	Generar un programa escrito en C++ que tenga una variable global constante haciendo uso de la palabra clave constexpr *const*.
Resultado esperado:	La variable global se ha borrado del fichero de código fuente.
Resultado obtenido:	Satisfactorio.

Tabla 52 - PR-11

Identificador: PR - 12	
Nombre:	Eliminación de variables globales constantes(constexpr)
Descripción:	Generar un programa escrito en C++ que tenga una variable global constante haciendo uso de la palabra clave *constexpr*.
Resultado esperado:	La variable global se mantiene inalterada en el código fuente.
Resultado obtenido:	Satisfactorio.

Tabla 53 - PR-12

Identificador: PR - 13	
Nombre:	Reescritura de variables globales no constantes
Descripción:	Generar un programa escrito en C++ que tenga una variable global inicializada no constante cuyo valor es modificado en el main.
Resultado esperado:	La variable global se ha reescrito en el *main* del programa.
Resultado obtenido:	Satisfactorio.

Tabla 54 - PR-13

Identificador: PR - 14	
Nombre:	Reescritura de variables globales constantes(const)
Descripción:	Generar un programa escrito en C++ que tenga una variable global constante haciendo uso de la palabra clave constexpr *const*.
Resultado esperado:	La variable global se ha reescrito en el *main* del programa.
Resultado obtenido:	Satisfactorio.

Tabla 55 - PR-14

Identificador: PR - 15	
Nombre:	Reescritura de variables globales constantes(constexpr)
Descripción:	Generar un programa escrito en C++ que tenga una variable global inicializada constante haciendo uso de la palabra clave *constexpr*.
Resultado esperado:	La variable global se mantiene inalterada en el código fuente.
Resultado obtenido:	Satisfactorio.

Tabla 56 - PR-15

Identificador: PR - 16	
Nombre:	Reescritura de variable global no constante como constexpr
Descripción:	Generar un programa escrito en C++ que tenga una variable global no constante inicializada con un valor conocido en tiempo de compilación, cuyo valor no es modificado en el resto del programa.
Resultado esperado:	La variable global se ha reescrito como *constexpr* en el mismo lugar en el que estaba definida.
Resultado obtenido:	Satisfactorio.

Tabla 57 - PR-16

Identificador: PR - 17	
Nombre:	Reescritura de variable global no constante como constante
Descripción:	Generar un programa escrito en C++ que tenga una variable global no constante inicializada con un valor desconocido en tiempo de compilación, cuyo valor no es modificado en el resto del programa.
Resultado esperado:	La variable global se ha reescrito como *const* en el main del programa.
Resultado obtenido:	Satisfactorio.

Tabla 58 - PR-17

Identificador: PR - 18	
Nombre:	No reescritura de variable global redefinida en el main
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global definida con un nombre “a”. • Definir una variable local en el main cuyo nombre es también “a”.
Resultado esperado:	La variable global no se ha borrado en el código fuente del programa.
Resultado obtenido:	Satisfactorio.

Tabla 59 - PR-18

Identificador: PR - 19	
Nombre:	Reescritura de funciones
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global inicializada • Una función que no tenga parámetros y que haga uso de la variable global. Llamar en el *main* del programa a la función.
Resultado esperado:	La función ha sido reescrita, teniendo un nuevo parámetro cuyo tipo y nombre es el mismo que el de la variable global.
Resultado obtenido:	Satisfactorio.

Tabla 60 - PR-19

Identificador: PR - 20	
Nombre:	Reescritura de funciones
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global inicializada • Una función que tenga parámetros y que haga uso de la variable global. Llamar en el *main* del programa a la función.
Resultado esperado:	La función ha sido reescrita, teniendo un nuevo parámetro cuyo tipo y nombre es el mismo que el de la variable global.
Resultado obtenido:	Satisfactorio.

Tabla 61 - PR-20

Identificador: PR - 21	
Nombre:	Reescritura de llamadas a función
Descripción:	Generar un programa escrito en C++ que tenga: • Una variable global inicializada • Una función que tenga parámetros y que haga uso de la variable global. Llamar en el *main* del programa a la función.
Resultado esperado:	La llamada a la función ha sido reescrita, teniendo un nuevo parámetro cuyo nombre es el mismo que el de la variable global.
Resultado obtenido:	Satisfactorio.

Tabla 62 - PR-21

Identificador: PR - 22	
Nombre:	Reescritura de constantes globales definidas mediante macros
Descripción:	Generar un programa escrito en C++ que tenga una constante global definida mediante una macro. Utilizar la constante en el main del programa.
Resultado esperado:	La constante global definida mediante macro ha sido sustituida por una variable global definida con *constexpr* y de tipo *auto*.
Resultado obtenido:	Satisfactorio.

Tabla 63 - PR-22

Identificador: PR - 23	
Nombre:	No reescritura de constantes globales definidas mediante macros
Descripción:	Generar un programa escrito en C++ que tenga una constante global definida mediante una macro. Esta constante estará definida dentro de un fragmento #ifdef - #endif. Utilizar la constante en el main del programa.
Resultado esperado:	La constante global definida mediante macro no se ha modificado en el código fuente.
Resultado obtenido:	Satisfactorio.

Tabla 64 - PR-23

6.2 Pruebas de integración

Una vez se superan las pruebas unitarias, es necesario probar el software en el conjunto de todas sus partes, como una sola unidad. Con este fin se realizan las pruebas de integración. Para satisfacer los requisitos, estas pruebas se realizarán con dos benchmarks de la suite de PARSEC. Los benchmarks elegidos son Blackscholes y Swaptions, ya que son programas con la complejidad suficiente como para tener un resultado satisfactorio. Para describir cada una de las pruebas realizadas se utilizará una tabla como la siguiente:

Identificador: PRI - XX	
Nombre:	Nombre de la prueba
Descripción:	Descripción de la prueba
Resultado esperado:	Resultado esperado
Resultado obtenido:	Satisfactorio o no satisfactorio.

Tabla 65 - Ejemplo tabla prueba de integración

A continuación, se describen las pruebas de integración:

Identificador: PRI - 01	
Nombre:	Eliminación de variables globales en Blackscholes
Descripción:	Para la realización de la prueba hay que llevar a cabo los siguientes pasos: • Descargar la suite PARSEC • Borrar del código fuente del programa Blackscholes el código referente a paralelismo ya que se realizará una ejecución secuencial. • Transformar el fichero de formato .c a formato .cpp. • Modificar el makefile con la última modificación. • Ejecutar la herramienta sobre el fichero fuente principal, bl*ackscholes.cpp.*
Resultado esperado:	La herramienta no realiza ninguna modificación, ya que en el código original existe la siguiente macro: *#define fptype float,* la cual es incompatible con la herramienta.
Resultado obtenido:	Satisfactorio.

Tabla 66 - PRI-01

Identificador: PRI - 02	
Nombre:	Eliminación de variables globales en Blackscholes
Descripción:	Para la realización de la prueba hay que llevar a cabo los siguientes pasos: • Descargar la suite PARSEC • Borrar del código fuente del programa Blackscholes el código referente a paralelismo ya que se realizará una ejecución secuencial. • Borrar la macro *#define fptype float.* • Sustituir en el código todos los elementos de tipo *fptype* por *float*. • Transformar el fichero de formato .c a formato .cpp. • Modificar el makefile con la última modificación. • Ejecutar la herramienta sobre el fichero fuente principal, bl*ackscholes.cpp.* • Ejecutar el benchmark de nuevo y comprobar que los resultados son los mismos que los obtenidos en una ejecución del código original.
Resultado esperado:	La herramienta modifica el código fuente borrando las variables globales y el resultado de ejecutar el benchmark con el código modificado es el mismo que al ejecutarlo con el código original.
Resultado obtenido:	Satisfactorio.

Tabla 67 - PRI-02

Identificador: PRI - 03	
Nombre:	Eliminación de variables globales en Swaptions
Descripción:	Para la realización de la prueba hay que llevar a cabo los siguientes pasos: • Descargar la suite PARSEC • Borrar del código fuente del programa Swaptions el código referente a paralelismo ya que se realizará una ejecución secuencial. • Ejecutar la herramienta sobre el fichero fuente principal, HJM_Securities*.cpp.*
Resultado esperado:	La herramienta no realiza ninguna modificación, ya que en el código original existe la siguiente macro: *#define fptype double,* la cual es incompatible con la herramienta.
Resultado obtenido:	Satisfactorio.

Tabla 68 - PRI-03

Identificador: PRI - 04	
Nombre:	Eliminación de variables globales en Swaptions
Descripción:	Para la realización de la prueba hay que llevar a cabo los siguientes pasos: • Descargar la suite PARSEC • Borrar del código fuente del programa Swaptions el código referente a paralelismo ya que se realizará una ejecución secuencial. • Borrar la macro *#define fptype double.* • Sustituir en el código todos los elementos de tipo *fptype* por *double*. • Ejecutar la herramienta sobre el fichero fuente principal, HJM_Securities*.cpp.* • Ejecutar el benchmark de nuevo y comprobar que los resultados son los mismos que los obtenidos en una ejecución del código original.
Resultado esperado:	La herramienta modifica el código fuente borrando las variables globales y el resultado de ejecutar el benchmark con el código modificado es el mismo que al ejecutarlo con el código original.
Resultado obtenido:	Satisfactorio.

Tabla 69 - PRI-04

6.3 Evaluación

A pesar de que mejorar el rendimiento de las aplicaciones no es un objetivo de este proyecto es conveniente realizar unas pequeñas y breves comparaciones para comprobar cómo afectan estas modificaciones sobre los programas en su rendimiento.

La evaluación se realizará sobre los dos benchmarks elegidos en las pruebas de integración, blackscholes y swaptions. Ambos se ejecutarán de forma secuencial pues son los programas tipo objetivos de esta herramienta. Ambas mediciones de tiempo se realizarán sobre el conjunto de datos simlarge, el mayor conjunto de datos para simulación proporcionado por PARSEC.

Blackscholes

Este benchmark cuenta con 11 variables globales, las cuales son eliminadas tras la ejecución de la herramienta. Existe una función que en el código original recibía un solo parámetro y en el código transformado recibe nueve más. Los resultados obtenidos en 15 ejecuciones del benchmark son los siguientes:

Input: SimLarge	Blackscholes original	Input: SimLarge	Blackscholes modificado	
Ejecución	Tiempo	Ejecución	Tiempo	SpeedUp
1	1,363	1	1,367	0,99707388
2	1,365	2	1,489	0,91672263
3	1,367	3	1,374	0,99490539
4	1,378	4	1,358	1,01472754
5	1,374	5	1,356	1,01327434
6	1,368	6	1,356	1,00884956
7	1,359	7	1,374	0,98908297
8	1,38	8	1,371	1,00656455
9	1,364	9	1,366	0,99853587
10	1,363	10	1,359	1,00294334
11	1,362	11	1,355	1,00516605
12	1,365	12	1,355	1,00738007
13	1,36	13	1,358	1,00147275
14	1,359	14	1,36	0,99926471
15	1,361	15	1,357	1,00294768
Media	**1,365866667**	**Media**	**1,370333333**	

Tabla 70 - Tiempos blackscholes

Los tiempos son bastante pequeños y muy parecidos en ambos casos. La diferencia en la media es de solamente 5 milésimas, lo que nos permite concluir que para este caso concreto el rendimiento del programa no ve alterado por estas modificaciones.

Swaptions

Este benchmark cuenta 8 variables globales, de las cuales 6 son eliminadas y definidas como variables locales en el main. Las otras dos se reescriben como expresiones constantes globales que serán resueltas en tiempo de compilación. Además, se transforman tres constantes definidas mediante macros en expresiones constantes globales. Los resultados obtenidos en 15 ejecuciones del benchmark son los siguientes:

Input: SimLarge	Swaptions original	Input: SimLarge	Swaptions modificado	
Ejecución	Tiempo	Ejecución	Tiempo	SpeedUp
1	6,391	1	6,369	1,00345423
2	6,376	2	6,374	1,00031377
3	6,373	3	6,357	1,00251691
4	6,406	4	6,364	1,00659962
5	6,377	5	6,362	1,00235775
6	6,371	6	6,362	1,00141465
7	6,395	7	6,359	1,00566127
8	6,367	8	6,366	1,00015708
9	6,398	9	6,372	1,00408035
10	6,375	10	6,365	1,00157109
11	6,377	11	6,361	1,00251533
12	6,371	12	6,368	1,00047111
13	6,455	13	6,336	1,01878157
14	6,371	14	6,383	0,99812001
15	6,373	15	6,359	1,0022016
Media	**6,385066667**	**Media**	**6,3638**	

Tabla 71 - Tiempos swaptions

En este caso el tiempo de ejecución es más notable, unas 5 veces mayor. En este caso, al igual que en el anterior, los tiempos son muy similares y las medias solo se diferencian en dos centésimas de segundo. En este caso concreto podemos concluir también que las transformaciones realizadas no afectan al rendimiento del programa.

En estos dos casos concretos, en el que el número de variables globales no es muy elevado, el rendimiento del programa se ve inalterado. A cambio, obtenemos un código cuya mantenibilidad y legibilidad es mucho mejor.

Cada programa es un problema diferente para la herramienta y por ello no podemos realizar ninguna afirmación genérica ya que no sabemos cómo afectaría al rendimiento sobre programas con varias decenas de variables globales o programas con múltiples funciones que hagan uso de bastantes variables globales.

6.4 Ejemplos de funcionamiento

En este apartado se mostrarán diferentes ejemplos de funcionamiento de la herramienta haciendo uso de pequeños ejemplos.

6.4.1 Ejemplo 1

```
1  int a = 1, b=2, c=3;
2  double dob = 3;
3  int copia = b;
4
5  void func(){
6
7  a++;
8  c++;
9
10 }
11
12 int main(){
13
14 func();
15 return 1;
16 }
```

Ilustración 11 - Ejemplo 1 Inicial

```
1  constexpr int b = 2;
2  constexpr double dob = 3;
3
4  void func ( int &a , int &c ){
5
6  a++;
7  c++;
8
9  }
10
11 int main(){
12
13
14 int a = 1;
15 int c = 3;
16 const int copia = b;
17 func(a , c );
18 return 1;
19 }
```

Ilustración 10 - Ejemplo 1 Final

En este ejemplo, tenemos dos variables globales que se modifican durante la ejecución del programa. Estas variables son *a* y *c*, las cuales se reescriben como variables locales del *main* sin alterar sus propiedades.

Por otro lado, las variables globales *copia*, *b* y *dob* no se modifican durante la ejecución del programa. Las variables *b* y *dob* tienen un valor conocido en tiempo de compilación, por lo que se declaran como expresiones constantes de ámbito global que serán resueltas por el compilador en tiempo de compilación. No son un problema en tiempo de ejecución. Sin embargo, la variable global copia, es una copia de b que no se modifica posteriormente, por lo que se declarará en el main de programa como una constante.

Finalmente tenemos que la función *func* hace uso de las variables globales *a* y *c* y, además, las modifica. Al pasarlas al main, tenemos que añadirlas como nuevos parámetros de la función *func.* En el resultado final se puede observar como *func* recibe las variables globales de las que hace uso como parámetro. Son pasadas por referencia puesto que originalmente las modificaba.

6.4.2 Ejemplo 2

```
int a = 1;
int b = 2;
void func(){
a++;
b++;
}

void pasarela(){

func();

}
int main(){

pasarela();

return 1;
}
```

Ilustración 13 - Ejemplo 2 Inicial

```
1
2 void func ( int &a , int &b ){
3 a++;
4 b++;
5 }
6
7 void pasarela ( int &a , int &b ){
8
9 func(a , b );
10
11 }
12 int main(){
13
14 int a = 1;
15 int b = 2;
16 pasarela(a , b );
17
18 return 1;
19 }
```

Ilustración 12- Ejemplo 2 Final

En este segundo ejemplo, se muestra un caso extremo en el que una función intermediaria llama a otra función que utiliza variables globales. Como se puede observar en el resultado final, las variables globales se declaran como locales en el main y ambas funciones reciben como nuevos parámetros la referencia de las variables globales de la que hacen uso. La función *func* es la que hace uso directo de estas variables mientras que la función pasarela simplemente llama a la función *func*. Al declarar como locales las variables *a* y *b*, es necesario que la función pasarela las reciba como parámetro para poder hacérselas llegar a la función *func* y así esta pueda manipularlas.

6.4.3 Ejemplo 3

```
1  #define macroconst 3
2
3  constexpr int b = 2;
4  const   double dob = 3;
5
6
7  int main(){
8
9  int a = macroconst;
10
11 return 1;
12 }
```

Ilustración 15 - Ejemplo 3 Inicial

```
1
2  constexpr auto macroconst = 3;
3
4  constexpr int b = 2;
5  constexpr   double dob = 3;
6
7
8  int main(){
9
10 int a = macroconst;
11
12 return 1;
13 }
```

Ilustración 14- Ejemplo 3 Final

En este tercer ejemplo se muestran dos casos concretos de transformaciones de constantes. El primero de ellos alude a las constantes definidas mediante macros. En el ejemplo inicial tenemos la constante *macroconst* definida mediante macro, al pasar la herramienta, esta se transforma en una expresión constante de tipo auto que el compilador se encargará de resolver.

El segundo caso es un caso de optimización de constantes. El usuario ha declarado una variable global, cuyo valor es conocido en tiempo de compilación, como constante. Al conocer su valor en tiempo de compilación, esta variable se puede declarar como una expresión constante en vez de como una simple constante. De esta forma, la variable global se resolverá en tiempo de compilación.

6.4.4 Ejemplo 4

```
1  class Rectangle {
2      int width, height;
3    public:
4      Rectangle(int w,int h){
5          width = w;
6          height = h;
7
8      }
9      int get_width() const {return width;}
10     void set_width(int a){width = a;}
11     int get_height() const {return height;}
12 };
13
14
15 Rectangle rec{2, 2};
16 int noinit;
17 int repetida = 1;
18
19
20 int main(){
21
22 int repetida = 2;
23 rec.get_width();
24 rec.set_width(8);
25
26 return 1;
27 }
```

Ilustración 16 - Ejemplo 4 Inicial

```
1  class Rectangle {
2      int width, height;
3    public:
4      Rectangle(int w,int h){
5          width = w;
6          height = h;
7
8      }
9      int get_width() const {return width;}
10     void set_width(int a){width = a;}
11     int get_height() const {return height;}
12 };
13
14
15 int repetida = 1;
16
17
18 int main(){
19
20 Rectangle rec{2, 2};
21 int noinit;
22 int repetida = 2;
23 rec.get_width();
24 rec.set_width(8);
25
26 return 1;
27 }
```

Ilustración 17 - Ejemplo 4 Final

En este ejemplo se muestran 3 casos diferentes. El primero de ellos consiste en mostrar un ejemplo diferente de transformación de variable global a variable local. Para ello en este caso se ha hecho uso de un objeto declarado de forma global. El objeto, un rectángulo, se inicializa en su creación y posteriormente se modifica en el main del programa. Por ello, se reescribe como variable local del main sin alterar sus propiedades.

El segundo caso visible en el ejemplo es la eliminación de una variable global no inicializada, que no si quiera se usa, pero que igual que las demás se elimina y se declara como local en el main.

Por último, tenemos un caso extremo en el que tenemos una variable global y una variable local en el main declaradas con el mismo nombre. En este caso no podemos eliminar la variable global ya que existe una variable con el mismo nombre en el main. Por ello, después de la transformación, sigue declarada como una variable global.

7. Conclusiones

Una vez finalizado el proyecto, es necesario comprobar si se han cumplido todos los objetivos planteados al comienzo del mismo. Asimismo, se plantean distintas opciones para trabajos futuros.

7.1 Cumplimiento de objetivos

Para ver si se han cumplido los objetivos, se verifica el cumplimiento o no de los distintos requisitos planteados.

Requisito	Resultado	Observaciones
RNF-01	Completado	
RNF-02	Completado	
RNF-03	Completado	
RNF-04	Completado	
RNF-05	Completado	
RNF-06	Completado	
RNF-07	Completado	
RNF-08	Completado	
RF-01	Completado	
RF-02	Completado	
RF-03	Completado	
RF-04	Completado	
RF-05	Completado	
RF-06	Completado	
RF-07	Completado	
RF-08	Completado	
RF-09	Completado	
RF-10	Completado	
RF-11	Completado	
RF-12	Completado	
RF-13	Completado	
RF-14	Completado	Le herramienta informa de la correcta ejecución o no de la herramienta. Se podría ofrecer más información al usuario.
RF-15	Completado	
RF-16	Completado	

Tabla 72 - Verificación del cumplimiento de requisitos

7.2 Líneas futuras

Todo proyecto software es mejorable y este, no es una excepción. Es posible plantear una serie de mejoras o alternativas para trabajos futuros respecto a la reescritura de código de forma automática y eliminación de variables globales.

Por un lado, en lo referente a la reescritura de código, la API de Clang es bastante potente pero aun así muy mejorable. La documentación proporcionada es escasa en un gran número de funciones y el desarrollador tiene que estar jugando a prueba y error.

Trabajar con macros resulta bastante complejo y la API necesita mejorar también en este apartado. Existe un caso ya comentado a resolver y es la incompatibilidad del API para la reescritura de elementos cuyo tipo de dato está definido mediante una macro *(#define ftype float*).

Por otro lado, en lo referente a la eliminación de variables globales, se puede plantear una implementación diferente cuyo objetivo secundario, además de la eliminación de variables globales, sea la optimización del rendimiento. Se podrían realizar transformaciones mucho más complejas como las descritas en [21], que seguro mejorarían el rendimiento del código resultante que ofrece esta implementación e incluso del código original del usuario.

Se podría ofrecer al usuario más flexibilidad dándole la oportunidad de transformar las variables que el desee, una a una. También se le puede ofrecer más información en tiempo de ejecución sobre las variables globales que tiene en el código.

Otra mejora menor sería la de ofrecer una interfaz al usuario ya que la herramienta está implementada para usar mediante la consola de comandos.

De todas estas opciones planteadas, sin duda, la más beneficiosa y útil sería la de eliminar las variables globales centrándose en la optimización del rendimiento del código.

8. Planificación

En este apartado se compara la planificación inicial establecida con los resultados finales de la duración del proyecto.

La planificación final dista mucho de la inicial por dos motivos principalmente. El primero de ellos y más notable son los parones por estudio. Uno de estos parones ocurre durante los meses de diciembre y enero. Este parón se debe a la finalización del primer cuatrimestre del curso y a la realización de los exámenes finales de enero. El otro parón notable ocurre a mitad de mayo, fecha de los exámenes del segundo cuatrimestre.

El segundo motivo de las grandes diferencias en la planificación es la inexperiencia a la hora de planificar y establecer periodos de tiempo en este tipo de proyectos.

A continuación, se muestran la planificación inicial y final del proyecto:

Planificación Inicial

Actividad	Fecha de inicio	Duración	Fecha de finalización
Propuesta del proyecto	01/10/2015	1	01/10/2015
Planificación	02/10/2015	2	03/10/2015
Estado del arte	04/10/2015	5	08/10/2015
Requisitos de usuario	09/10/2015	2	10/10/2015
Casos de uso	11/10/2015	1	11/10/2015
Requisitos software	12/10/2015	2	13/10/2015
Arquitectura del sistema	15/10/2015	3	16/10/2015
Especificación entorno tecnológico	17/10/2015	1	17/10/2015
Decisiones de diseño	18/10/2015	3	20/10/2015
Instalación y configuración	21/10/2105	5	25/10/2015
Aprendizaje y documentación	26/10/2015	15	09/11/2015
Implementación MyFrontendAction	10/11/2015	1	10/11/2015
Implementación MyASTConsumer	11/11/2015	1	11/11/2015
Implementación Find_Includes	12/11/2015	3	14/11/2015
Implementación MyASTVisitor	15/11/2015	60	13/01/2016
Pruebas Unitarias	14/01/2016	5	18/01/2016
Pruebas de Integración	19/01/2016	2	20/01/2016
Conclusiones	21/01/2016	1	21/01/2016
Presupuesto	22/01/2016	1	22/01/2016

Tabla 73 - Planificación Inicial

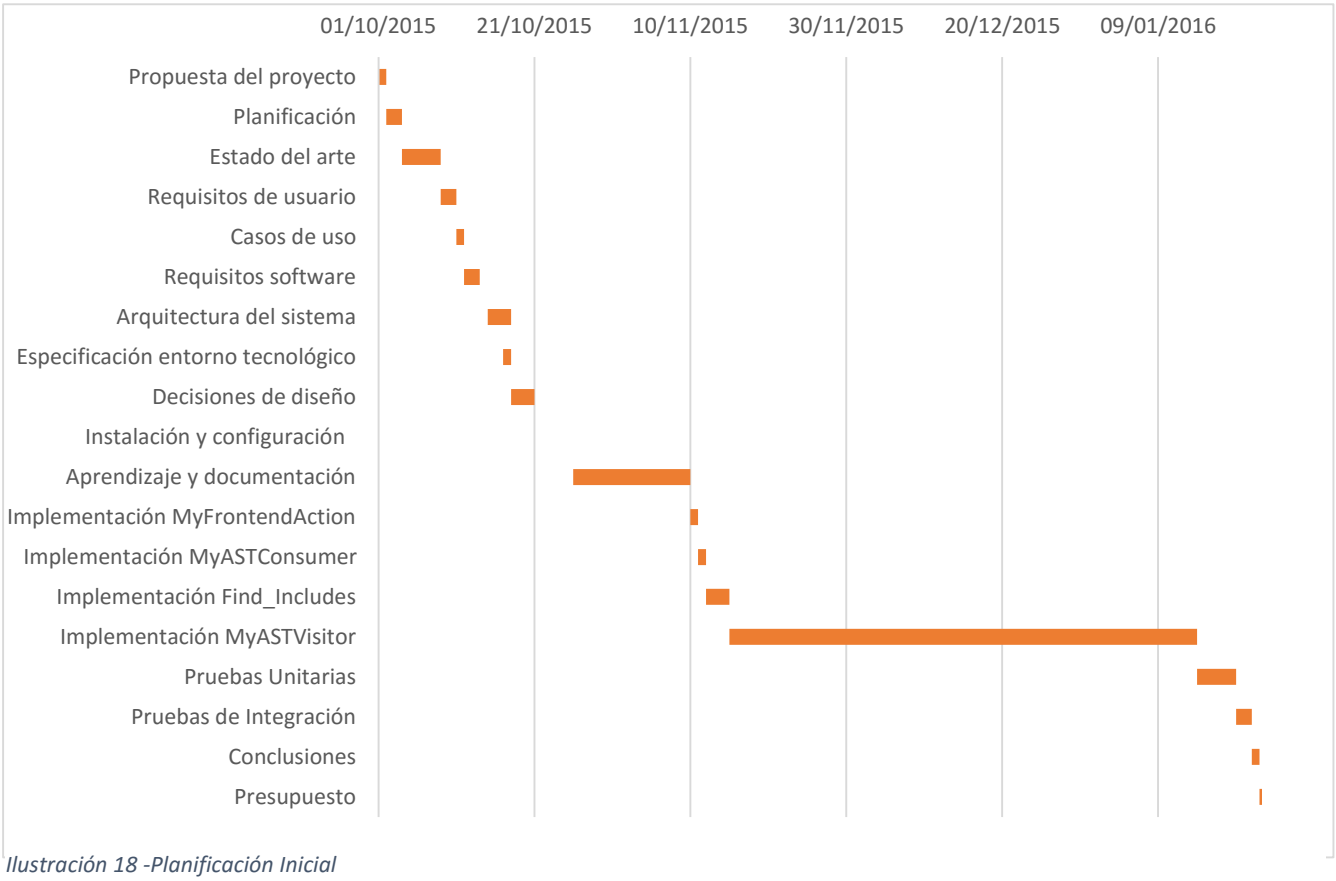

Ilustración 18 -Planificación Inicial

Actividad	Fecha de inicio	Duración	Fecha de finalización
Propuesta del proyecto	01/10/2015	1	01/10/2015
Planificación	02/10/2015	2	03/10/2015
Estado del arte	06/10/2015	7	12/10/2015
Requisitos de usuario	14/10/2015	6	19/10/2015
Casos de uso	20/10/2015	1	20/10/2015
Requisitos software	24/10/2015	7	30/10/2015
Arquitectura del sistema	03/11/2015	10	12/11/2015
Especificación entorno tecnológico	14/11/2015	1	14/11/2015
Decisiones de diseño	16/11/2015	3	18/11/2015
Instalación y configuración	20/01/2006	3	22/01/2006
Aprendizaje y documentación	26/01/2016	9	03/02/2016
Implementación MyFrontendAction	04/02/2016	1	04/02/2016
Implementación MyASTConsumer	08/02/2016	1	08/02/2016
Implementación Find_Includes	09/03/2016	3	11/03/2016
Implementación MyASTVisitor	12/03/2016	56	06/05/2016
Pruebas Unitarias	09/05/2016	5	13/05/2016
Pruebas de Integración	14/05/2016	2	15/05/2016
Conclusiones	18/05/2016	1	18/05/2016
Presupuesto	25/05/2016	1	25/05/2016

Tabla 74 - Planificación final

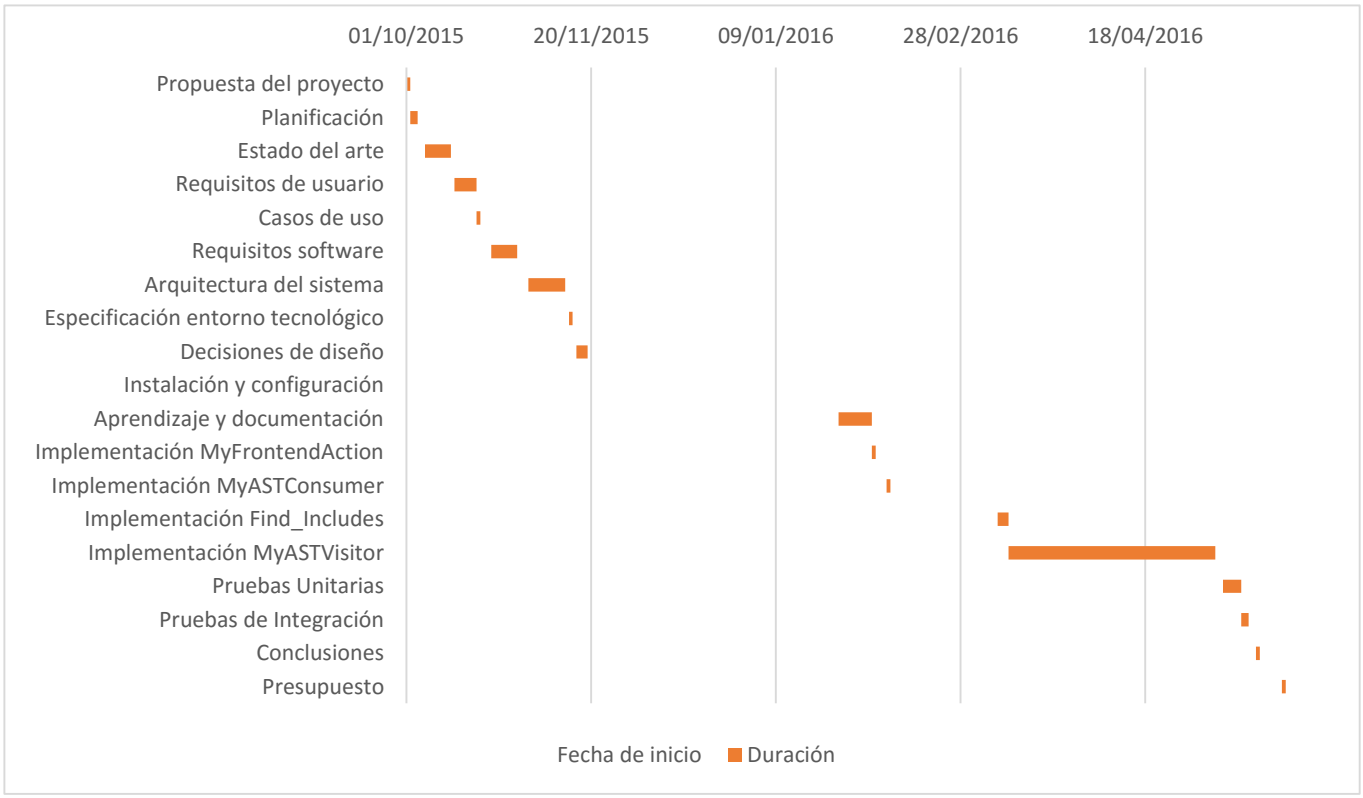

Ilustración 19 - Planificación final

9. Presupuesto

El presupuesto sin IVA para la realización del proyecto será de doce mil setecientos treinta y un euros con cincuenta y cinco céntimos (12.731,55 €), a lo que hay que añadir el IVA, que actualmente es de un 21%. El desglose de los costes se detalla a continuación:

Personal

Puesto	Horas	Coste(€/hora)	Coste total(€)
Jefe de proyecto	50	40	2000
Programador	495	20	9900
		Total (€)	11900

Tabla 75 - Coste personal

Equipos

Descripción	Coste de cada equipo	Coste total de los equipos	Meses de uso	Coste aplicable al proyecto
1 Ordenador Portátil	499 €	499 €	8 meses	166,67 €
			Total (€)	166,67

Tabla 76 - Coste equipos

Software

Descripción	Coste	Meses de uso	Coste aplicable al proyecto
Microsoft Office 2016	149 €	8 meses	49,67 €
		Total (€)	49,67

Tabla 77 - Coste software

Material fungible

Descripción	Coste de cada elemento	Coste total de los elementos
1 paquetes Papel Din A-4	2,95 €	2,95 €
2 bolígrafos	0,5 €	1 €
1 Paquetes 1000 Post - it	5 €	5 €
	Total (€)	8,95

Tabla 78 - Coste material fungible

Otros gastos

Meses de uso	Coste aplicable al proyecto
Costes indirectos	5% de los costes directos
Total (€)	606,26 €

Tabla 79 - Otros gastos

Resumen

Concepto	Coste (€)
Personal	11900
Equipos	166,67
Software	49,67
Material fungible	8,95
Otros gastos	606,26
Total sin IVA (€)	12.731,55 €
Total con IVA (€)	15.405,17 €

Tabla 80 - Resumen de costes

10. Marco Regulador

Este apartado se dedica al aspecto legal de todo lo referente a este proyecto. Se detallarán los distintos estándares o licencias de aquellas tecnologías utilizadas en el proyecto.

10.1 C++

En lo referente a C++, el lenguaje utilizado para el desarrollo de la herramienta, el proyecto se cimienta sobre dos estándares concretos: ISO/IEC 14882:2011 y su revisión ISO/IEC 14882:2014. Ambos especifican los requisitos para la implementación del lenguaje de programación C++. C++ es un lenguaje de programación de propósito general basado en el lenguaje de programación C, tal y como se describe en especificación ISO/IEC 9899:1999. [32]

10.2 Parsec suite

PARSEC benchmark suite tiene Copyright (c) 2006-2009 Princeton University. Su redistribución y uso, con o sin modificación del mismo, está permitida siempre y cuando se cumplan las siguientes condiciones [33]:

- La redistribución del código fuente debe contener el copyright indicado, esta lista de condiciones y la renuncia de responsabilidad.
- La redistribución de forma binaria debe contener el copyright indicado, esta lista de condiciones y la renuncia de responsabilidad.
- Ni el nombre de la Universidad de Priceton ni el nombre de los desarrolladores puede usarse para promocionar productos derivados de este software sin el permiso correspondiente.

Renuncia de responsabilidad:

"THIS SOFTWARE IS PROVIDED BY PRINCETON UNIVERSITY ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PRINCETON UNIVERSITY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."

10.3 LLVM-Clang

La versión de LLVM utilizada tiene Copyright (c) 2003-2015 University of Illinois at Urbana-Champaign mientras que Clang tiene Copyright (c) 2007-2015 University of Illinois at Urbana-Champaign. Cualquier persona con una copia del software puede hacer uso del mismo sin restricciones ni limitaciones en cuanto a uso, copia, modificación, publicación, distribución, introducción de sub-licencias o ventas siempre y cuando se respeten las siguientes condiciones:

- La redistribución del código fuente debe contener el copyright indicado, esta lista de condiciones y la renuncia de responsabilidad.
- La redistribución de forma binaria debe contener el copyright indicado, esta lista de condiciones y la renuncia de responsabilidad.
- Ni el nombre de la Universidad de Illinois en Urbana-Champaign, ni el nombre del equipo de LLVM, ni el nombre de los desarrolladores puede usarse para promocionar productos derivados de este software sin el permiso correspondiente.

Renuncia de responsabilidad:

“THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.”

11. Referencias

[1]B. Stroustrup, *C++ programming language*. *4th ed*, 2013.

[2]"A Tour of C++", *isocpp*, 2016. [Online]. Available: https://isocpp.org/tour. [Accessed: 12- May- 2016].

[3]"C++11 FAQ", *Stroustrup.com*, 2016. [Online]. Available: http://www.stroustrup.com/C++11FAQ.html. [Accessed: 12- May- 2016].

[4]"C++11 Language Extensions — General Features", *https://isocpp.org*, 2016. [Online]. Available: https://isocpp.org/wiki/faq/cpp11-language. [Accessed: 12- May- 2016].

[5]"C++11 Standard Library Extensions — Containers and Algorithms", *https://isocpp.org*, 2016. [Online]. Available: https://isocpp.org/wiki/faq/cpp11-library-stl. [Accessed: 12- May- 2016].

[6]"C++14 Language Extensions", *https://isocpp.org*, 2016. [Online]. Available: https://isocpp.org/wiki/faq/cpp14-language. [Accessed: 12- May- 2016].

[7]"C++11 Overview", *https://isocpp.org*, 2016. [Online]. Available: https://isocpp.org/wiki/faq/cpp14. [Accessed: 12- May- 2016].

[8]"The LLVM Compiler Infrastructure Project", *Llvm.org*, 2016. [Online]. Available: http://www.llvm.org/. [Accessed: 08- May- 2016].

[9]""clang" C Language Family Frontend for LLVM", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/. [Accessed: 12- May- 2016].

[10]"Clang - Features and Goals", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/features.html. [Accessed: 10- May- 2016].

[11]"Choosing the Right Interface for Your Application — Clang 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/docs/Tooling.html. [Accessed: 10- May- 2016].

[12]"Overview — Clang 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/docs/ClangTools.html. [Accessed: 12- May- 2016].

[13]"ClangCheck — Clang 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/docs/ClangCheck.html. [Accessed: 12- May- 2016].

[14]"Clang-Tidy — Extra Clang Tools 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/. [Accessed: 18- Jun- 2016].

[15]"clang-tidy - modernize-loop-convert — Extra Clang Tools 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/checks/modernize-loop-convert.html. [Accessed: 18- Jun- 2016].

[16]"clang-tidy - modernize-use-nullptr — Extra Clang Tools 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/checks/modernize-use-nullptr.html. [Accessed: 18- Jun- 2016].

[17]"clang-tidy - modernize-use-auto — Extra Clang Tools 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/checks/modernize-use-auto.html. [Accessed: 18- Jun- 2016].

[18]"clang-tidy - Clang-Tidy Checks — Extra Clang Tools 3.9 documentation", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/checks/list.html. [Accessed: 18- Jun- 2016].

[19] D. Binkley, M. Harman, Y. Hassoun, S. Islam and Z. Li, "Assessing the impact of global variables on program dependence and dependence clusters", *Journal of Systems and Software*, vol. 83, no. 1, pp. 96-107, 2010.

[20] Gengbin Zheng, Stas Negara, Celso L. Mendes, Laxmikant V. Kalé, "Automatic Handling of Global Variables for Multi-threaded MPI Programs".

[21] J. Daniel Garcia, Bjarne Stroustrup, "Improving performance and maintainability through refactoring in C++11", August 27, 2015.

[22] H. Sankaranarayanan and P. Kulkarni, "Source-to-Source Refactoring and Elimination of Global Variables in C Programs", JSEA, vol. 06, no. 05, pp. 264-273, 2013.

[23]"The PARSEC Benchmark Suite", *Parsec.cs.princeton.edu*, 2016. [Online]. Available: http://parsec.cs.princeton.edu/overview.htm. [Accessed: 18- Jun- 2016].

[24]"The PARSEC Benchmark Suite", ARCO Group seminar 27th January of 2012, 2016.

[25]C.Bienia,[Online].Available: http://courses.cs.washington.edu/courses/cse471/10sp/hw/bienia08characterization_p3_p4.pdf. [Accessed: 18- Jun- 2016].

[26]"Spoon - Source Code Analysis and Transformation for Java | Spoon", *Spoon.gforge.inria.fr*, 2016. [Online]. Available: http://spoon.gforge.inria.fr/. [Accessed: 18- Jun- 2016].

[27]"clang: clang::PPCallbacks Class Reference", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html. [Accessed: 18- Jun- 2016].

[28]"clang: clang::RecursiveASTVisitor< Derived > Class Template Reference", *Clang.llvm.org*, 2016.[Online].Available: http://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html. [Accessed: 18- Jun- 2016].

[29]"clang: clang::Rewriter Class Reference", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/doxygen/classclang_1_1Rewriter.html. [Accessed: 18- Jun- 2016].

[30]"clang: clang::ASTConsumer Class Reference", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html. [Accessed: 18- Jun- 2016].

[31]"clang: clang::ASTFrontendAction Class Reference", *Clang.llvm.org*, 2016. [Online]. Available: http://clang.llvm.org/doxygen/classclang_1_1ASTFrontendAction.html. [Accessed: 18- Jun- 2016].

[32]"ISO/IEC 14882:2014 - Information technology -- Programming languages -- C++", *ISO*, 2016. [Online].Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029. [Accessed: 18- Jun- 2016].

[33]"The PARSEC Benchmark Suite", *Parsec.cs.princeton.edu*, 2016. [Online]. Available: http://parsec.cs.princeton.edu/license.htm. [Accessed: 18- Jun- 2016].

Anexo A: Summary

A.1 Introduction

Global variables are one of the most basic parts that a programming language has and they are included in almost every programming language. A global variable is a variable that can be access from every other single part of the program because its scope is global.

Global variables are very easy to use and they make the life easier for developers because they make developers to write less lines of code and to develop faster. When programmers make use of global variables, is usually because there is some kind of data that must be accessed from different scopes or functions. As they are declared in a global scope, every function of the program can access it without having to pass it as a function parameter. This is the main reason that makes programmers to use a lot global variables, leaving apart other possible reasons for its use such us concurrency or performance.

In spite of this noticeable benefit or ease of the global variables, an excessive use of them without any control of what is being done it's not recommended. In some programming languages, such as C++, its use is not recommended. Some of the reasons that makes global variables not being recommended are:

- Hurts maintainability of source code.
- Makes code more difficult to read and harder to understand.
- Makes the process of debugging harder.
- They introduce a higher probability of making developing mistakes.
- They are alive in memory until the end of the execution.

Is noticeable that global variables aren't good enough for using them freely and without thinking on its consequences but its abusive use is widely spread among developers.

Changing developer's way of programming it's not easy as it's not easy to change the customs of the people. Programmers usually develop their programmes with the most comfortable and easy tools they have. If a new change in a programming language makes coding easier and takes shorter time of the programmer, this change is going to be adopted and used among developers which use this language. This is not that easy with global variables because globals are something basic in programming, that can't be modified easily and developers have been using them for a very long time ago.

In order to change this abusive use of global variables, is necessary to provide developers something that is easy and fast to use. This are the reasons why we thought in developing a tool that could change developers source code automatically, eliminating global variables.

A.2 Objectives

The main objective of this project is to develop a tool capable of transform existing global variables in a C++ program to local variables declared in the main function of the program. This includes the refactoring of functions, introducing as new parameters the global variables used by that functions.

The tool, apart from eliminating global variable, will also perform as a constant transformer. The tool will be capable of declaring as constant a global variable that wasn't declared originally as constant, if it satisfies the requirements for being a constant.

To test the correct implementation of the tool, we have selected two programs that must work correctly after being transformed by the tool. This two programs are two benchmarks of PARSEC suite which are real and great programs for testing our tool

In order to develop our tool, it's necessary to use some kind of libraries that allow us to obtain all the information from the source files. The library that is will be used in the project development is called LibTooling and it belongs to Clang libraries. It will allow us to analyse any C++ source code and develop, using C++ programming language, our tool for eliminating global variables.

A.3 Clang Libtooling

For developing our tool or any other tool which modifies source code, we need a library that allow us to do it. There are no many choices here. We have Spoon [26], an open source library which allows developers to analyse source code and modify it. It's not useful for us because is for Java programming language.

Another option is using one of the three different interfaces that Clang offers. These ones are made for C and C++ programming languages, therefore, they are valid for us. As it has been said, Clang offers three different options for source code analysis and rewriting. Each one of those three options has some specific characteristics:

- **LibClang:** is an interface developed in C programming language with a high abstraction level. It's recommended for tools which do not make use of C++ programming language characteristics or when there is no need of having full control over the AST. Recommended when you are in doubt. [11]

- **Clang Plugins:** this interface allows developers to write plugins in order to do additional actions with the AST during compilation process. They are not recommended for standalone tools. [11]

- **LibTooling:** this interface is written in C++ and its objective is to ease the developing of standalone tools and its integration with clang services. It's recommended when full control of the AST is needed, when you want to work with C++ source files or when the tool will work with more than one source code file. [11]

Libclang is not valid because is written in C and we need a C++ interface for getting all the necessary information of the source code files.

The plugin option is also discard because we want to develop a standalone tool.

Libtooling is the only choice that fits with the requirements defined by the client and fits our needs due to it is written in C++ and it's going to allow us to obtain all the source code information we need. In addition to this, it allows us to create a standalone tool and we can manipulate the AST more freely, which means less restrictions and more possibilities. Because all of this reasons, our choice for developing the tool is Clang LibTooling.

A.4 Analysis

As it has been said before, the objective of this project is to develop a tool for eliminating global variables from C++ programs by modifying the source code. The main functionalities gathered from our client are the following:

- Identify global variables.
- Identify constants defined with macros.
- Classify global variables as constants or not constants.
- Remove global variables.
- Rewrite global variables in the main function of the program.
- Transformation of non-constant global variables to constants, if they can be defined as constants.
- Modify function definitions and function calls, adding as parameters those global variables that were used inside the function before they were removed.
- It won't modify source code if any error is detected.

These functionalities must be formally detailed in the user requirements specification and must be satisfied by one or more software requirements.

Other characteristics

All the transformation process it will be done by a single tool and in one execution of the tool.

The source code to change will be passed as parameter in the tool execution command, passing the main program file route. Not only the main program file will be changed, all files in the same directory will be treated.

No copies of the source code files will be created during the execution; all modifications will be done over the original source code files.

The user has to be sure that his program compiles without any errors, otherwise the results of the execution can be altered.

A.5 Implementation

After the analysis and design phases the next step is the implementation of the tool. We must fulfil all the requirements described in the software analysis.

Before starting the developing process, it's mandatory to download and install LLVM-Clang. Also will be necessary git, for downloading LLVM-Clang, and cmake, for building LLVM-Clang. The general steps before starting to develop the tool are the followings:

- Download and install git.
- Download and install cmake.
- Download llvm with git.
- Download clang with git.
- Download compiler-rt with git.
- Download clang tools extra with git.
- Build and install LLVM-Clang with cmake.

After this installing and configure steps we can now start developing our tool with Clang libraries. Here is a brief description of each class implemented in the tool:

Find_Includes class

This class inherits from PPCallbacks class from Clang libraries [27]. PPCallbacks provides a set of functions that, in case of being implemented, are executed if a specific condition happens. This functions are focused in preprocessor actions and allow us to locate and get information of every macro used in the source code. It is going to be used for getting the location and information of constants defined by macro directives and *#ifdef/#endif* o *#ifndef/#endif* macros.

MyASTVisitor class

This class inherits from RecursiveASTVisitor class from Clang libraries [28]. RecursiveASTVisitor class main purpose is to traverse every single AST node given in clang AST, allowing us to obtain all the information needed for global variables elimination and rewriting.

This class is the largest one and it does the most complex things. It is in charge of obtaining the relevant information of the user source code, classifying it, removing global variables and rewriting global variables.

This class can be divided in two parts, the first one is about gathering all the necessary information for doing the rewriting and eliminating process. RecursiveASTVisitor has different functions that will be executed when a AST node matches a function

characteristic. With some of this offered functions we0ll gather all the information needed. The second part is developed in a proper function. This function will be in charge of processing all the information gathered and removing and rewriting the global variables and functions based in this collected information.

MyASTConsumer class

This class inherits from ASTConsumer class from Clang libraries [30]. ASTConsumer class allows us to access the AST. This class is in charge of the initialization of a MyASTVisitor object for traversing the AST nodes.

MyFrontendAction class

This class inherits from ASTFrontendAction class from Clang libraries [31]. This class allows us to work with the AST of a source code file. For each file creates a MyASTConsumer object.

This class will also allow us to initialize the Find_Include class in order to gather preprocessor operations information.

This four classes make the whole tool. The tool must be compiled with the clang libraries linked getting as a result an executable program that must be executed through the command line of a terminal. It will receive a single parameter which is the complete directory where the main file of the program to be treated is located.

A.6 Results and evaluation

Tests

After de implementation is finished, is mandatory to test the final result in order to make sure that all the requirements described in the analysis part are fulfilled and the tool works correctly. In order to verify this, it's necessary to design and describe a wide range of tests.

Twenty three unit test have been designed in order to test each functionality of the tool. All of this tests have been passed by the developed tool.

When each functionality of the tool is tested individually, is necessary to test the whole tool as a single unit. Integration tests are the proper tests for this. For this integration tests, we have chosen 2 big programs included in PARSEC benchmark suite.

PARSEC benchmark [23] suite is a group of thirteen multithreaded benchmarks focused on the evaluation of multicore shared memory processors.

Each of this benchmarks have different input data with different size of data, from the lowest quantity of data possible, which can take milliseconds of execution, to real problems data, which can take several minutes of execution.

For the integrity tests two benchmarks have been selected out of the thirteen:

- **Blackscholes: "**This application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation" [25].
- **Swaptions: "**The application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions" [25].

This two benchmarks are big programs enough for the integrity tests. Four integrity tests have been designed with this two programs, two tests for each one. The four tests have been passed by the tool developed.

All the tests designed have been passed and all the mandatory requirements have been fulfilled. We can say that the results have been positive as all the objectives have been achieved.

Perfomance

In spite of improving programs performance is not the objective of this project, is always a good practice to test if the modifications done in the source code by the tool affects in some way the performance of the program.

This little evaluation will be done over the two benchmarks selected for the integration tests, blackscholes and swaptions. Both of them will be executed in a secuential way because sequential programs are the target programs of this tool.

Both time measurements will be done with simlarge data input offered by PARSEC suite.

Blackscholes

This benchmark has eleven global variables and all of them are eliminated after the tool execution. All of them are transformed to local variables in the main function of the program. There is one function in the program that in the original implementation only received one parameter, after the tool modification it received nine new parameters. The results of fifteen different executions are the following:

Input: SimLarge Execution	Original blackscholes Time	Input: SimLarge Execution	Modified blackscholes Time
1	1,363	1	1,367
2	1,365	2	1,489
3	1,367	3	1,374
4	1,378	4	1,358
5	1,374	5	1,356
6	1,368	6	1,356
7	1,359	7	1,374
8	1,38	8	1,371
9	1,364	9	1,366
10	1,363	10	1,359
11	1,362	11	1,355
12	1,365	12	1,355
13	1,36	13	1,358
14	1,359	14	1,36
15	1,361	15	1,357
Mean	**1,365866667**	**Mean**	**1,370333333**

Tabla 81 - Blackscholes timing

AS it can be seen in the table, all times are very similar and small. The difference between the two means is of five thousandths of seconds which is almost no noticeable. We can say that for this example, the performance is not affected by the changes that have been done by the tool.

Swaptions

This benchmark has eight global variables, six of them have been transformed to main local variables and the other two have been transformed to *constexpr* global variables which will be solved in compilation time. It's also necessary to mention that 3 constants defined with macros have been transformed to constexpr global variables which also

will be solved in compilation time. The results of fifteen different executions are the following:

Input: SimLarge Execution	Original swaptions Time	Input: SimLarge Execution	Modified swaptions Time
1	6,391	1	6,369
2	6,376	2	6,374
3	6,373	3	6,357
4	6,406	4	6,364
5	6,377	5	6,362
6	6,371	6	6,362
7	6,395	7	6,359
8	6,367	8	6,366
9	6,398	9	6,372
10	6,375	10	6,365
11	6,377	11	6,361
12	6,371	12	6,368
13	6,455	13	6,336
14	6,371	14	6,383
15	6,373	15	6,359
Mean	**6,385066667**	**Mean**	**6,3638**

Tabla 82 - Swaptions timing

In this case, execution time is around five times bigger but it doesn't make it different from the blackscholes case. Again, all the times are very similar and the means just differs one from the other in hundredths of second, in favour of the modified version. The difference is not significant and it only can be said that the changes done by the tool, again, do not affect the performance of the program.

For this two examples in which the number of global variables wasn't too big, the performance of both programs isn't harmed but neither improved.

A.7 Examples

A.7.1 Example 1

```
1  int a = 1, b=2, c=3;
2  double dob = 3;
3  int copia = b;
4
5  void func(){
6
7  a++;
8  c++;
9
10 }
11
12 int main(){
13
14 func();
15 return 1;
16 }
```

Ilustración 21 - Example 1 Initial code

```
1  constexpr int b = 2;
2  constexpr double dob = 3;
3
4  void func ( int &a , int &c ){
5
6  a++;
7  c++;
8
9  }
10
11 int main(){
12
13
14 int a = 1;
15 int c = 3;
16 const int copia = b;
17 func(a , c );
18 return 1;
19 }
```

Ilustración 20 - Example 1 final code

In this example, we can see two global variables which are modified during the program execution. These two global variables are *a* and *c*, which are rewrited in the main function of the program without changing anything of them.

There are also three global variables which aren't modified during the execution, therefore, they should be constant. The value of *b* and *dob* global variables is known in compilation time and they can be rewrited as global constant expressions which will be resolved by the compiler at compilation time. In the other hand, *copia* is a copy of *b*, that's why this global variable is rewrited as constant in the main function of the program.

The last change made by the tool Is passing by reference to the function func the rewrited global variables that it makes use of.

A.7.2 Example 2

```
int a = 1;
int b = 2;
void func(){
a++;
b++;
}

void pasarela(){

func();

}
int main(){

pasarela();

return 1;
}
```

Ilustración 22 – Example 2 initial code

```
1
2 void func ( int &a , int &b ){
3 a++;
4 b++;
5 }
6
7 void pasarela ( int &a , int &b ){
8
9 func(a , b );
10
11 }
12 int main(){
13
14 int a = 1;
15 int b = 2;
16 pasarela(a , b );
17
18 return 1;
19 }
```

Ilustración 23- Example 2 final code

In this second example we can see a special case in which one function calls another function which uses two global variables. The two global variables are rewrited as local variables in the main function. Both variables must be modified by func, so they must be passed as reference to func. As function p*asarela* calls *func*, it must pass these variables to it as reference and for doing it, *pasarela* must receive them also as reference. So, both functions must receive as reference both global variables and that's what it can be observed in the final code picture.

A.7.3 Example 3

```
1  #define macroconst 3
2
3  constexpr int b = 2;
4  const   double dob = 3;
5
6
7  int main(){
8
9  int a = macroconst;
10
11 return 1;
12 }
```

Ilustración 25 - Example 3 Initial code

```
1
2  constexpr auto macroconst = 3;
3
4  constexpr int b = 2;
5  constexpr   double dob = 3;
6
7
8  int main(){
9
10 int a = macroconst;
11
12 return 1;
13 }
```

Ilustración 24- Example 3 final code

This third example shows two different transformations. The first one is a macro defined constant transformation to a global constant expression with auto type, which will be solved by the compiler at compile time.

The second transformation is an optimization. Global variable *dob* is declared as constant but it's value is known at compile time so it can be declared as a constant expression rather than a simple constant. As it can be observed in the final code picture, *dob* has been transformed to a global constant expression which will be solved at compilation time.

A.7.4 Exmaple 4

```
1  class Rectangle {
2      int width, height;
3    public:
4      Rectangle(int w,int h){
5          width = w;
6          height = h;
7
8      }
9      int get_width() const {return width;}
10     void set_width(int a){width = a;}
11     int get_height() const {return height;}
12 };
13
14
15 Rectangle rec{2, 2};
16 int noinit;
17 int repetida = 1;
18
19
20 int main(){
21
22 int repetida = 2;
23 rec.get_width();
24 rec.set_width(8);
25
26 return 1;
27 }
```

Ilustración 26 – Example 4 initial code

```
1  class Rectangle {
2      int width, height;
3    public:
4      Rectangle(int w,int h){
5          width = w;
6          height = h;
7
8      }
9      int get_width() const {return width;}
10     void set_width(int a){width = a;}
11     int get_height() const {return height;}
12 };
13
14
15 int repetida = 1;
16
17
18 int main(){
19
20 Rectangle rec{2, 2};
21 int noinit;
22 int repetida = 2;
23 rec.get_width();
24 rec.set_width(8);
25
26 return 1;
27 }
```

Ilustración 27 –Example 4 final code

In this last example 3 different cases are shown. The first one is an object type global variable called rec which is eliminated and rewrited in the main function as a local variable.

The second one is the rewrite of a non-initialized global variable to the main function.

The last case is a special case in which the same variable is declared with a global scope and in the main function scope. Because of this, this global variable can't be eliminated, that's why it remains as it was originally in the final code.

A.8 Conclusions

At the end of the project we can say that we have accomplish our objectives. We have developed a tool that transform global variables into main local variables from C++ programs and all the unit testing has been passed and also the integration tests.

It's true that the project has suffered a big delay in the planning's but this wasn't a problem because we had time enough. In fact, the problem wasn't the delay but the planning. My inexperience in this kind of projects made me to design a bad planning with a bad day estimation.

I would like to talk about the Clang libraries used for developing the tool. Clang offers a very nice library full of useful functions for working with source code files. Is a powerful API but still have a big improvement margin. From my point of view, the API is very complete but lacks of documentation. Most of the functions aren't explained and there are no examples of use in any of them. Working with the macros and all relative to the preprocessor is also a bit hard and could be improved.

This developed tool has fulfilled our objectives but still can be improved to be much better and complete. One of the most significant improvements for this tool could be the transformations based on improving the performance of the program. Instead of doing the changes that the implemented tool does, an improved version of this tool can do the changes to global variables mentioned in [21], which are much more complex and would have much better performance.

Other minor improvements would be orientated to give the user more possibilities and facilities, for example:

- Giving the user the possibility to change only the selected global variables.
- Giving more feedback to the user during and after the execution.
- Developing a user interface.

As said, this is not the perfect tool but is very good start point for automatically source code transformation tools.