Clang

Un compilador de código abierto

Pedro Delgado Pérez
pedro.delgado@uca.es

Índice

1. Introducción a Clang

Clang es un compilador de código abierto para los lenguajes de la familia C: C, C++, Objective-C y Objective-C++[1] [1]. Se trata de un compilador reciente, cuyo desarrollo comenzó en 2005 y su primera versión con licencia de código abierto se produjo en 2007. La versión actual de Clang es la 3.7, lanzada en septiembre de 2015, aunque el desarrollo es bastante activo y salen nuevas versiones del producto con cierta frecuencia. El origen de este compilador, que surge del proyecto LLVM, será comentado en §2. En este documento, se emplea la versión 3.6 para sistema Ubuntu 14.04, aunque también se darán algunas indicaciones respecto a las versión 3.4 cuando sea necesario.

Clang está escrito en C++ y puede ser utilizado en sistemas operativos basados en UNIX. El compilador más conocido y usado para estos lenguajes ha sido y sigue siendo GCC (GNU Compiler Collection), desarrollado por el proyecto GNU para dar soporte a diversos lenguajes de programación. Sin embargo, existe una gran variedad de compiladores para estos y otros lenguajes (http://en.wikipedia.org/wiki/List_of_compilers). Clang a día de hoy puede ser usado como una alternativa directa al compilador GCC, ofreciéndonos ciertas ventajas, entre las que destaca su diseño modular para el análisis estático del código. Estas ventajas serán enumeradas en §3, y el análisis estático basado en el árbol de sintaxis abstracta será abordado en §6 y §7.

Para facilitar su uso a usuarios acostumbrados a GCC, el proyecto Clang ha buscado que el frontend sea altamente compatible con este compilador. De esta manera, podemos encontrar que el uso por línea de comandos es muy similar y que muchas de las opciones incluidas son compartidas, como se verá más adelante en §5. Para completar el documento, se mostrará la instalación de la herramienta en §4 y se darán unos apuntes finales sobre el análisis estático de código en §8.

2. Proyecto LLVM

Para entender el origen de Clang, primero es necesario conocer la existencia del proyecto LLVM. LLVM es un proyecto de código abierto que busca la creación de compiladores para cualquier lenguaje de programación, proporcionando la infraestructura necesaria para su desarrollo. No obstante, hay que destacar que, inicialmente, la idea del proyecto era centrarse en los lenguajes C y C++. Dentro de un compilador, podemos distinguir entre *frontend* y *backend*: el frontend traduce el código fuente en una representación intermedia, la cual es entendible por el backend, que convierte finalmente esta representación en código máquina. LLVM, como backend, provee de las capas intermedias para conformar un sistema de compilación completo, tomando la representación interna generada por el frontend creado para un determinado lenguaje de programación.

LLVM, siguiendo en la línea de compatibilidad con GCC, puede ser configurado para ser usado con este compilador a fin de poder disponer de un frontend y poder tratar cualquiera de los lenguajes que GCC incluye[2]. Es decir, que se

[1] En este documento nos centramos en los lenguajes C y C++.

[2] Aunque GCC inicialmente se diseñó para compilar lenguaje C, posteriormente se extendió para abarcar también a C++. Más adelante se desarrollaron frontends para otros lenguajes, que pueden consultarse en http://gcc.gnu.org/frontends.html

puede utilizar GCC como frontend para un determinado lenguaje, de forma que se emplee LLVM como backend para constituir una herramienta completa. Sin embargo, existen varios subproyectos en el proyecto LLVM para la construcción de nuevos frontends para diferentes lenguajes que trabajen de forma específica sobre LLVM (http://llvm.org/ProjectsWithLLVM), tal como Ruby, Python o PHP. Muchos de estos proyectos han atraído la atención para su uso tanto en desarrollos comerciales como de código abierto.

Clang es uno de los subproyectos originados del proyecto LLVM y que se dedica a la familia de lenguajes de C. Se trata de uno de los proyectos más consolidados, ya que incluso se distribuye con las versiones de LLVM, y, como proyecto de código abierto, es posible utilizar las bibliotecas que emplea. Clang forma parte de las versiones que lanza LLVM desde su versión 2.6 en octubre de 2009, lo que demuestra la madurez e importancia del mismo.

3. Ventajas de Clang

Como se comentó en §1, una de las mayores ventajas de Clang es el diseño modular de su desarrollo, que parece ser uno de los pilares desde la concepción del proyecto. Esto nos permite disponer de una API muy bien diseñada y poder emplearla para la construcción de nuestras propias herramientas para el análisis del código, de la misma manera en la que el frontend lo hace al compilarlo. En otras palabras, como proyecto de código abierto, podemos reutilizar las bibliotecas que nos ofrece el proyecto Clang para embeber de forma sencilla el compilador en las nuevas aplicaciones que estemos desarrollando, lo cual no es tan sencillo en otros compiladores como GCC. Entre las actividades que podemos emprender gracias a esto, podemos mencionar el análisis estático, la refactorización o la generación de código.

A continuación, se enumeran algunas de las ventajas que el proyecto Clang proclama:

- Uno de sus objetivos es reducir el tiempo de compilación y el uso de la memoria. Su arquitectura puede permitir de forma más directa el crear perfiles del coste de cada capa. Además, cuanta menos memoria toma el código, mayor cantidad del mismo se podrá incluir en memoria al mismo tiempo, lo cual beneficia el análisis del código.

- Informa de errores y avisos de una manera muy expresiva para que sea lo más útil posible, indicando exactamente el punto en el que se produce el error e información relacionada con el mismo.

- Clang busca que las herramientas desarrolladas puedan integrarse fácilmente con Entornos de Desarrollado Integrados (IDEs), a fin de que su ámbito de aplicación se amplíe.

- Utiliza la licencia BSD, la cual permite que Clang sea empleado en productos comerciales.

- Clang trata de ajustarse de la forma más completa posible a los estándares de los lenguajes de la familia C y sus variantes. Para aquellas extensiones soportadas que, de alguna manera no concuerdan de manera oficial con el estándar, son emitidas como avisos para conocimiento del desarrollador.

Por otra parte, podemos comentar las siguientes ventajas de Clang sobre GCC:

- En cuanto a la velocidad de compilación y uso de memoria, se ha verificado, de forma independiente a Clang, que el tiempo de compilación es normalmente mejor que usando GCC[3]. Sin embargo, el tiempo de ejecución es mejor con GCC, aunque Clang ha medrado mucho en este aspecto en los últimos años, y se espera una mejora aún mayor gracias a la actividad constante en su desarrollo.
- La licencia BSD es más permisiva que la GPL de GCC, lo cual no permite embeber la herramienta en software que no esté licenciado como GPL.
- Se hacen cumplir más las normas de los lenguajes. Por ejemplo, no se permite algo como '`void f(int a, int a);`'.
- En cuanto a los mensajes de error y aviso, Clang proporciona bastante más información, como la columna exacta del error, el rango afectado e incluso sugerencias de mejora[4].
- Se incluye soporte para un mayor número de extensiones del lenguaje y hereda características útiles de LLVM como backend (por ejemplo, soporte para la optimización del tiempo de enlazado).

Por último, y no menos importante, Clang dispone de documentación y tiene soporte mediante una lista de correos activa en la que se puede encontrar ayuda a los problemas con el uso de la herramienta.

4. Instalación

En la siguiente página, http://clang.llvm.org/get_started.html, aparecen instrucciones para comenzar con Clang. Como el compilador forma parte de las versiones de LLVM, lo mejor es acceder a http://llvm.org/releases/ para obtener la última versión. En caso de que quieras involucrarte en el desarrollo del proyecto, puedes seguir las instrucciones del apartado "Building Clang and Working with the Code" para obtener el código, y también acceder a la página http://clang.llvm.org/get_involved.html para saber cómo actuar en este caso.

Para el enlazado del código mostrado en §7, es posible que necesite instalar las siguientes librerías a fin de evitar errores comunes con el enlazador *ld* de GNU:

```
sudo apt-get install zlib1g-dev libtinfo-dev libedit-dev
```

Si surgen problemas durante la instalación, puedes encontrar ayuda en http://llvm.org/bugs/.

[3] http://en.wikipedia.org/wiki/Clang#Performance_and_GCC_compatibility
[4] http://clang.llvm.org/diagnostics.html

4.1. Instalación manual

Una vez dentro de la página de las versiones de LLVM (http://llvm.org/releases/), pincha en "download" de la versión a descargar. De la zona "Pre-built binaries", elige la que se ajuste a tu sistema y haz click sobre la misma para iniciar la descarga del archivo con formato "tar.xz". Como en este documento se emplea la versión 3.6 para un sistema Ubuntu 14.04, nos descargaremos el fichero con nombre `clang+llvm-3.6.0-x86_64-linux-gnu-ubuntu-14.04.tar.xz` pinchando en "Clang for Ubuntu 14.04".

Una vez descargado el fichero en la carpeta de "Descargas", basta con ejecutar lo siguiente desde la línea de comandos para descomprimir el archivo:

```
cd /usr/local/
sudo tar -xJf ~/Descargas/clang+llvm-3.6.0-x86_64-linux \
-gnu-ubuntu-14.04.tar.xz --strip 1
```

El resultado debe ser que cada fichero vaya a su correspondiente carpeta `bin`, `lib`, `man`...

Ahora, desde cualquier directorio, se debe poder ejecutar '`clang`' y '`clang++`'[5] sin problemas, por ejemplo, para comprobar la versión:

```
clang --version
clang++ --version
```

4.2. Instalación con paquetes APT

En la página http://llvm.org/apt/, en la zona "Install (stable branch)" se pueden observar los paquetes necesarios para la instalación de Clang. Para poder hacer uso de Clang y sus librerías, ejecutaremos el siguiente comando que instalará los paquetes necesarios:

```
sudo apt-get install clang-3.6 libclang-3.6-dev
```

Al hacer esta instalación, los ejecutables contendrán el nombre de la versión, es decir, que en lugar de '`clang`' y '`clang++`' tendremos que ejecutar '`clang-3.6`' y '`clang++-3.6`'. Por esta razón es recomendable realizar los siguientes enlaces simbólicos:

```
sudo ln -s /usr/bin/clang-3.6 /usr/bin/clang
sudo ln -s /usr/bin/clang++-3.6 /usr/bin/clang++
sudo ln -s /usr/bin/llvm-config-3.6 /usr/bin/llvm-config
```

5. Opciones

El formato de uso de Clang es el siguiente:

`clang/clang++` [*opción...*] [*entrada ...*]

[5] En caso de no disponer en el sistema de los paquetes esenciales para programar en C o C++, puedes obtenerlos tras instalar el paquete `build-essential` mediante la orden `sudo apt-get install build-essential`

Se puede utilizar `clang` o `clang++` dependiendo de la librería estándar con la que se desea enlazar la entrada, del mismo modo que en GCC existen las órdenes `gcc` y `g++`[6]. Como puede observarse, se puede seleccionar más de una opción y entrada. A continuación, se listan algunas de las opciones de uso más común al emplear Clang y que pueden ser más útiles para el manejo básico de la herramienta:

-E Solo ejecuta el preprocesado.

-S Solo ejecuta los pasos de preprocesado y compilación.

-c Solo ejecuta los pasos de preprocesado, compilación y ensamblado.

-o <***fichero***> Escribe la salida a un fichero.

-I <***directorio***> Añade una ruta para la búsqueda de las cabeceras.

-isystem <***directorio***> Añade un directorio para la búsqueda de cabeceras del sistema.

-g Genera información para *debug* del código fuente.

-pg Permite instrumentación para la creación de perfiles.

-std=<***valor***> Nos permite seleccionar el estándar del lenguaje que queremos que sea empleado para el procesado de código. Por ejemplo, si utilizamos `--std=c++11`, el estándar C++ 11 será utilizado [2], pudiendo distinguir las características adicionales de este estándar.

-stdlib=<***valor***> Librería estándar de C++ a emplear.

-w Elimina los mensajes de avisos.

-W<***aviso***> Activa un aviso específico. Con `-Weverything` activas todos los diagnósticos. Con `-Werror`, los avisos se transforman en errores.

-x <***lenguaje***> Trata los ficheros de entrada como si fuesen del lenguaje especificado. Esto quiere decir que esta opción tiene prioridad sobre usar `clang` o `clang++`.

La lista completa de opciones puede mostrarse con el comando `clang --help`. También se puede obtener más información en `http://linux.die.net/man/1/clang`.

6. Árbol de sintaxis abstracta

Durante este tutorial, hemos nombrado varias veces el término *frontend* y también *backend* y de cómo la conexión entre ambos se basa en que este último toma la representación intermedia creada por el primero. Pero, ¿qué es esa representación intermedia? El frontend toma como entrada una cadena de caracteres plana, que es nuestro código a alto nivel. Tras los correspondientes

[6] Si se utiliza '`clang++`', se enlazará tanto con la librería estándar de C++ como la de C.

análisis léxico, sintáctico y semántico, transforma el código en una representación estructurada del programa que tiene forma de árbol: se trata del **Árbol de Sintaxis Abstracta**, conocido como **AST**.

A diferencia de un árbol de sintaxis concreta, un AST no contiene cada token del código explícitamente en el árbol, sino que las expresiones están representadas a través de la composición de ramas del árbol. Esto nos da una visión más clara de la estructura completa del código y nos permite realizar un análisis en profundidad del mismo, lo cual sería difícil de conseguir sin él. En este sentido, el AST facilita la búsqueda de aquellos nodos del árbol que cumplen los criterios de lo que se desea localizar.

Abajo se comentan las ventajas que Clang difunde sobre el AST que genera:

- El AST creado es fácilmente comprensible, incluso para aquellos que, aún conociendo el lenguaje, no están familiarizados con la forma en la que un compilador funciona internamente. Su formato, parecido a un documento XML, permite un proceso de aprendizaje más rápido y nos puede proporcionar un conocimiento más profundo de los casos que deberían ser localizados por la búsqueda concreta que realicemos sobre el mismo.

- Para cada token, Clang monitoriza información sobre dónde fue escrito o dónde fue expandido si estaba envuelto en una macro. Esto quiere decir que se muestra el rango de código real que cada nodo abarca, lo que permite un mapeo sencillo y rápido de árbol a código nuevamente.

- No hay simplificación de código en ningún momento. Por ejemplo, una simplificación del código podría darse cuando existen paréntesis que no son necesarios, o cuando en el código, tenemos la expresión `x-x` que podría ser directamente interpretado como `0` (tal y como hace GCC por ejemplo). Esto facilita el análisis y la refactorización del código.

- El AST puede ser serializado en un formato estructurado y ser guardado en disco, lo que permite que después pueda ser procesado por otro programa.

El AST puede ser un gran aliado para lograr comprender la manera en la que el compilador va a interpretar nuestro código. Por ejemplo, pongamos el caso en el que en una misma expresión se emplean varios operadores entre los cuales queremos verificar la prioridad. Por ejemplo, la siguiente expresión involucra a los operadores $=$, $+$ y $\times$:

$$a = b + c \times d$$

Para obtener el AST del fichero de código fuente que contiene esa expresión (pongamos que está incluida en un fichero llamado `ejemplo.cpp`), podríamos obtenerlo por pantalla de la siguiente manera:

```
clang++ -Xclang -ast-dump -fsyntax-only ejemplo.cpp
```

También es posible pasar la salida de la ejecución a un fichero (por ejemplo, `ast_ejemplo_auxiliar.txt`) y eliminar el código de colores para poder procesarlo mediante un editor de texto:

```
sed -r "s/\x1B\\[([0-9]{1,2}(;[0-9]{1,2})*)?[mGK]//g" \
ast_ejemplo_auxiliar.txt > ast_ejemplo_final.txt
```

```
| -BinaryOperator 0x778a560 <line:7:2, col:14> 'int' lvalue '='
|  |-DeclRefExpr 0x778a428 <col:2> 'int' lvalue Var 0x778a1c0
|                                                  'a' 'int'
|  `-BinaryOperator 0x778a538 <col:6, col:14> 'int' '+'
|    |-ImplicitCastExpr 0x778a520 <col:6> 'int' <LValueToRValue>
|    | `-DeclRefExpr 0x778a450 <col:6> 'int' lvalue Var 0x778a250
|                                                  'b' 'int'
|    `-BinaryOperator 0x778a4f8 <col:10, col:14> 'int' '*'
|      |-ImplicitCastExpr 0x778a4c8 <col:10> 'int' <LValueToRValue>
|      | `-DeclRefExpr 0x778a478 <col:10> 'int' lvalue Var 0x778a2e0
|                                                  'c' 'int'
|      `-ImplicitCastExpr 0x778a4e0 <col:14> 'int' <LValueToRValue>
|        `-DeclRefExpr 0x778a4a0 <col:14> 'int' lvalue Var 0x778a370
|                                                  'd' 'int'
```

El fragmento de AST asociado a la expresión arriba mencionada es el siguiente:

Como puede observarse, la propia estructura del árbol nos permite conocer la prioridad de ejecución de los operadores. El nodo `BinaryOperator` que representa al operador de multiplicación está más interno que el de suma, lo que indica que será necesario realizar antes la operación de multiplicación. De la misma manera, la última operación a realizar sería la de asignación. Esto nos da una muestra de cómo el AST nos informa de la manera en la que el compilador evalúa nuestro código.

Otra información que puede extraerse del AST anterior son los atributos `line` y `col`. El primero indica la línea en la que se encuentra la expresión, mientras el segundo sirve para indicar la posición o el rango que ocupa un elemento dentro de una línea de código. Esto permite el mapeo directo entre el AST y el código como se comentó anteriormente.

En el siguiente enlace puede una introducción más amplia al AST de Clang (http://clang.llvm.org/docs/IntroductionToTheClangAST.html). A lo largo de la próxima sección veremos qué otras utilidades puede tener el poder disponer del AST y emplear las librerías de Clang para el análisis estático del código.

7. Análisis estático con Clang

Imaginemos el siguiente caso:

Necesito conocer el número de clases que he definido en un código en C++.

¿Cómo podrías conseguir esta información? Una de las opciones es mirar visualmente el código e ir contando las mismas. Otra opción es usar un sistema de expresiones regulares que busque coincidencias con la palabra `class`. No obstante, ambas son propensas a errores. La primera por la propia intervención humana, y la segunda porque podríamos encontrarnos con ciertas dificultades: ¿y si por ejemplo la palabra `class` estuviera en un comentario? Este es un ejemplo que puede ser solventado con expresiones regulares a pesar de no estar al tanto de la semántica del lenguaje, pero podemos encontrarnos con la necesidad de analizar casos más complejos. ¿Es posible automatizar estas búsquedas para

encontrar exactamente lo que necesitamos? Es aquí donde entra en juego el AST y las bibliotecas que nos proporciona Clang.

En el AST, cada elemento del código que cubre el compilador es representado con un tipo de nodo diferente, que en la API de Clang esta representado por una clase concreta con cuyos métodos podemos extraer información del mismo. En el ejemplo que nos incumbe, una declaración de una clase es representada por la clase `CXXRecordDecl`. En la documentación de la API de Clang pueden verse cada una de las clases existentes[7]. Uno de los métodos que nos ofrece Clang es utilizar el patrón visitante para poder visitar todos los nodos que encuentre en el árbol de este tipo:

```
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/Tooling.h"
#include "clang/Tooling/CommonOptionsParser.h"

using namespace clang::tooling;
using namespace llvm;
using namespace clang;

class FindNumberClassesVisitor
  : public RecursiveASTVisitor<FindNumberClassesVisitor>{
public:
  explicit FindNumberClassesVisitor(ASTContext *Context)
    : Context(Context) { counter = 0; }

  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration){
    if(Declaration->isThisDeclarationADefinition()){
        llvm::outs() << "Clase : "
             << Declaration->getNameAsString() << "\n";
        FullSourceLoc FullLocation =
          Context->getFullLoc(Declaration->getLocStart());

        if( FullLocation.isValid()
            && !Context->getSourceManager().
                isInSystemHeader(FullLocation) )
          counter++;
    }
    return true;
  }
  unsigned int getCounter(){ return counter; }

private:
  ASTContext *Context;
  unsigned int counter;
};
```

[7] http://clang.llvm.org/doxygen/

```
class FindNumberClassesConsumer: public clang::ASTConsumer{
public:
  explicit FindNumberClassesConsumer(ASTContext *Context)
    : Visitor(Context) {}

  virtual void HandleTranslationUnit(
                      clang::ASTContext &Context){
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
    llvm::outs() << "El total de clases es: "
                 << Visitor.getCounter() << "\n";
  }
private:
  FindNumberClassesVisitor Visitor;
};

class FindNumberClassesAction:
        public clang::ASTFrontendAction{
public:
  virtual std::unique_ptr<ASTConsumer> CreateASTConsumer(
  clang::CompilerInstance &Compiler, llvm::StringRef InFile){
    return std::unique_ptr<clang::ASTConsumer>(
        new FindNumberClassesConsumer(
                        &Compiler.getASTContext()));
  }
};

static llvm::cl::OptionCategory OptC("opciones");

int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, OptC);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());
  return Tool.run(newFrontendActionFactory
                    <FindNumberClassesAction>().get());
}
```

Como se puede ver en el código de arriba[8], el método `VisitCXXRecordDecl` es el que implementa el patrón visitante para clases; por cada una de las clases encontradas, este método será invocado y dentro podemos llevar un contador que nos permita finalmente calcular el número de clases presentes en el código que han sido definidas. Como puede observarse, la clase que contiene al método `VisitCXXRecordDecl` (`FindNumberClassesVisitor`) hereda de la clase `RecursiveASTVisitor`. El objeto `Visitor` que se ha creado de la clase `FindNumberClassesVisitor`, llama al método `TraverseDecl` para comenzar el recorrido por el árbol. Una vez finalizado el recorrido, se puede mostrar el valor del contador. La utilidad del resto de clases y métodos del código anterior pueden ser encontradas en http://clang.llvm.org/docs/RAVFrontendAction.html.

[8]Este código ha sido validado en la versión 3.4 de Clang, pero es posible que algún método varíe en versiones posteriores, por lo que sería necesario adaptar el código a los cambios introducidos.

Version 3.4 Este código no es compatible hacia atrás con la versión 3.4 de Clang debido a varios cambios que se realizaron en las librerías en la versión 3.5. Los cambios a realizar son los siguientes:

- *Cambiar:*

```
class FindNumberClassesAction :
        public clang::ASTFrontendAction{
public:
   virtual std::unique_ptr<ASTConsumer> CreateASTConsumer(
   clang::CompilerInstance &Compiler, llvm::StringRef InFile){
      return std::unique_ptr<clang::ASTConsumer>(
              new FindNumberClassesConsumer(
                    &Compiler.getASTContext()));
   }
};
```

por:

```
class FindNumberClassesAction :
        public clang::ASTFrontendAction{
public:
   virtual clang::ASTConsumer *CreateASTConsumer(
   clang::CompilerInstance &Compiler, llvm::StringRef InFile){
      return new FindNumberClassesConsumer(
                           &Compiler.getASTContext());
   }
};
```

- *Cambiar:*

```
CommonOptionsParser OptionsParser(argc, argv, OptC);
```

por:

```
CommonOptionsParser OptionsParser(argc, argv);
```

- *Cambiar:*

```
return Tool.run(newFrontendActionFactory
          <FindNumberClassesAction>().get());
```

por:

```
return Tool.run(newFrontendActionFactory
          <FindNumberClassesAction>());
```

Para la obtención del ejecutable, se puede emplear el siguiente *makefile*, en el cual se enlaza con las librerías de Clang:

```
CXX := clang++
RTTIFLAG := -fno-rtti
CXXFLAGS := $(shell llvm-config --cxxflags) $(RTTIFLAG)
LLVMLDFLAGS := $(shell llvm-config --ldflags --libs --system-libs)

SOURCES = clases_file.cpp
OBJECTS = $(SOURCES:.cpp=.o)
EXES = $(OBJECTS:.o=)
CLANGLIBS = \
        -lclangFrontend \
        -lclangSerialization \
        -lclangDriver \
        -lclangTooling \
        -lclangParse \
        -lclangSema \
        -lclangAnalysis \
        -lclangEdit \
        -lclangAST \
        -lclangASTMatchers \
        -lclangLex \
        -lclangBasic \
        -lclangRewrite

clases_file: clases_file.o
        $(CXX) -o $@ $^ $(CLANGLIBS) $(LLVMLDFLAGS)
```

Version 3.4 Para la versión 3.4, realizar los siguientes cambios en el *makefile*:

- *Cambiar:*

  ```
  LLVMLDFLAGS := $(shell llvm-config --ldflags --libs \
                        --system-libs)
  ```

  por:

  ```
  LLVMLDFLAGS := $(shell llvm-config --ldflags --libs) \
                        -ldl
  ```

- *Cambiar:*

  ```
  -lclangRewrite
  ```

  por:

  ```
  -lclangRewriteCore \
  -lclangRewriteFrontend
  ```

Una vez creado el ejecutable a través de nuestro *makefile*, podemos emplearlo sobre un fichero de código de la siguiente forma:

```
./clases_file ejemplo_clases.cpp --
```

Podemos copiar nuestro programa al mismo directorio en el que está instalado Clang a fin de que ambas herramientas localicen las mismas cabeceras:

```
cp clases_file $(dirname $(which clang))
```

Una vez hecho esto, el programa será accesible desde otros directorios y podremos ejecutar directamente:

```
clases_file ejemplo_clases.cpp --
```

Si nos fijamos en el comando, este termina en `--`; justo tras estos guiones podemos indicar opciones a Clang (vea §5) si fuese necesario, por ejemplo, para señalar que hay una cabecera que ha de ser buscada en otro directorio (vea §8 para conocer una alternativa a este sistema).

Si el fichero `ejemplo_clases.cpp` es el siguiente:

```
class A;

class A {};

class B {};

class C;
```

La salida obtenida será:

```
Clase : A
Clase : B
El total de clases es: 2
```

Es posible que en nuestro código existan declaraciones previas de una clase, es decir, declaraciones de clases que son definidas posteriormente. La condición `Declaration->isThisDeclarationADefinition()` evita que estas declaraciones previas sean omitidas del cálculo, permitiendo encontrar solo las clases definidas. Es el caso de la clase C, que no está definida y por ello no se contabiliza. Por su parte, la clase A, que tenía declaración previa, se contabiliza una vez.

Una de las cosas que hay que notar es que en el AST estará presente todo el código de las cabeceras que se incluyan también. Esto implica por igual a las cabeceras desarrolladas por nosotros mismos como las cabeceras del sistema. En este sentido, la búsqueda también se realizará en las cabeceras a no ser que le indiquemos al compilador lo contrario. De ahí que se obtenga la localización de la clase en el código (se guarda en en el objeto `FullLocation` de la clase `FullSourceLoc`) y se compruebe la condición `!Context->getSourceManager().isInSystemHeader(FullLocation)`, para que si el elemento está en una cabecera del sistema no lo trate. Si queremos limitar la búsqueda al fichero que se procesa, sin incluir ciertas cabeceras, se puede emplear la opción `-isystem` (vea §5) para que trate las cabeceras que se indiquen junto a la opción como cabeceras del sistema.

8. Notas finales sobre el análisis estático

- En primer lugar, en el ejemplo de la sección anterior hemos vista una búsqueda concreta sobre un único fichero. Sin embargo, hay que comentar que es posible ejecutar de una misma vez una búsqueda sobre varios ficheros. Cada fichero será procesado individualmente, generando un AST para cada uno, pero se puede ir almacenando información global sobre los ficheros que se han ido estudiando. Pero además, también es posible realizar diversas comprobaciones en el mismo recorrido del AST. Por ejemplo, se podría realizar una búsqueda sobre declaraciones de métodos, para la cual añadiríamos un método `VisitCXXMethodDecl`.

- Clang nos proporciona otro método para la realización del análisis estático además de con la utilización del patrón visitante sobre cada clase concreta. Se trata de un lenguaje de diseño específico (DSL), conocido como *matchers*, que nos permite realizar una búsqueda de patrones sobre el árbol, localizando de forma directa los nodos que cumplen los requerimientos del patrón. Más información sobre esto puede ser encontrada en http://clang.llvm.org/docs/LibASTMatchers.html.

- Con las bibiliotecas de Clang también es posible realizar cambios en el código fuente, lo que se denomina como transformaciones de código a código [3]. Con ello, por ejemplo, se podrán realizar actividades de refactorización. Esto se consigue gracias al mapeo del AST con el código (comentado en §6), y también a la clase `Rewriter`, que nos permite realizar cambios en el código en la posición requerida. En los siguientes enlaces [4, 5] pueden encontrarse tutoriales en inglés sobre Clang y el uso de bibliotecas para el análisis del código.

- Clang nos proporciona un mecanismo para indicar el comando de compilación completo de cada uno de los ficheros de código fuente de entrada. Se trata de una base de datos de compilación denominada *JSON*. Información sobre cómo puede ser creada esta base de datos de compilación puede encontrarse en http://clang.llvm.org/docs/JSONCompilationDatabase.html. En este caso, se omite `--` al final de comando y Clang automáticamente buscara un fichero llamado `compile_commands.json` que deberá haber sido creado previamente.

- Por último, en la siguiente tesis [6] se puede comprobar la diversidad de acciones que se pueden llevar a cabo mediante análisis estático para mejorar la calidad del código. También es interesante el punto "2.2 Practical Issues"; en él se hace mención de la existencia de *falsos positivos* y *falsos negativos*, lo cual nos ha de hacer ser muy cuidadosos con nuestras búsquedas. Conocer las diversas posibilidades que deben llevar a la localización de un nodo y el propio estudio del AST son claves para un mayor refinamiento de la búsqueda.

Referencias

[1] Clang: a C language family frontend for LLVM clang.llvm.org Acceso: 04/11/2014. 2

[2] ISO. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. International Organization for Standardization, Ginebra, Suiza, Febrero 2012. 6

[3] Olaf Krzikalla. Performing source-to-source transformations with Clang. En *2013 European LLVM Conference*, Paris, Francia, Abril 2013. 14

[4] Kevin A. Boos. Clang Tutorial `http://kevinaboos.wordpress.com/2013/07/23/clang-tutorial-part-i-introduction/` Acceso: 04/11/2014. 14

[5] Larry Olson. A collection of code samples showing usage of Clang and LLVM as a library `https://github.com/loarabia/Clang-tutorial/` Acceso: 04/11/2014. 14

[6] Elias Penttilä. Improving C++ software quality with static code analysis. Tesis de Master, Aalto University, School of Science, Mayo 2014. 14

Este documento se escribió con Emacs, procesó con LaTeX 2_ε, y convirtió a PDF mediante `pdflatex`. La versión de Clang y LLVM empleada ha sido la 3.6, en el sistema GNU/Ubuntu Linux 14.04.

Se han usado los tipos Latin Modern.

Se terminó en Cádiz, en noviembre de 2015.