Nombre: John Alejandro González
Código: 1013336184

Nombre: Luis Alejandro Baena
Código: 1023628877

UNIVERSIDAD EAFIT

Lenguajes Formales y Automatas (CM0081)
Laboratorio III

Resumen

En este laboratorio se discute sobre la verificación de programas en representación intermedia de LLVM, el cual es un lenguaje de bajo nivel que se utiliza para representar programas de lenguajes de alto nivel, es un lenguaje que permite a los compiladores realizar optimizaciones de código.
Se utiliza la herramienta formal *VeLLVM* creada por *DeepSpec* para verificar programas en representación intermedia de LLVM.
Se discute la motivación del uso de las herramientas mencionadas, en los lenguajes mencionados.
Se discute la posibilidad de verificar programas que planteen problemas de teoría de números, ecuaciones diferenciales y grafos.
Se discute la posibilidad de revisar y estudiar formalmente sobre la implementación de verificación de programas hechos en "Brainf**ck" usando Interaction Trees.

Índice

1. Motivación **2**

2. Detalles Técnicos y versionamiento **2**
2.1. Sistema Operativo y versiones 2

3. Instalación de herramientas **2**

4. Verificación de programas en representación intermedia de LLVM **3**

5. Programas de interés **3**

6. Programas de futuro interes **3**

7. Verificación de programas en Brainfk** **4**

1. Motivación

Al momento de considerar la idea de verificar un programa, nos planteamos la posibilidad de verificar código en ensamblador. Inicialmente, supusimos que sería lógico comenzar con el lenguaje más bajo, y por ende, optamos por ensamblador. Sin embargo, al reflexionar más a fondo, especialmente al tener en cuenta que nuestro equipo es un 'Macbook Pro M2' con una arquitectura ARM, la cual difiere significativamente de la arquitectura x86 de Intel que conocemos, la idea de verificar ensamblador nos pareció 'fuera de alcance'. Consideramos las múltiples arquitecturas de computadoras existentes, cada una con sus propias instrucciones y diferencias, lo cual complicaría el proceso. Esta línea de pensamiento también nos llevó a contemplar la verificación de estructuras como 'FPGA (Field Programmable Gate Array)' o 'ASIC (Application Specific Integrated Circuit)', ya que habíamos trabajado con ellas en el pasado. Sin embargo, al igual que con la idea de verificar ensamblador, encontramos desafíos significativos.

Al replantear nuestra perspectiva, recordamos un video de "MIT OpenCourseWare"titulado "5. C to Assembly", donde se aborda la traducción del lenguaje C a ensamblador. En este video, se menciona que el compilador çlangütiliza un lenguaje intermedio denominado "LLVM". Este último es un lenguaje de bajo nivel diseñado para representar programas de lenguajes de alto nivel, permitiendo a los compiladores realizar optimizaciones de código.

La idea cobró fuerza cuando nos percatamos de que múltiples lenguajes de programación, como Rust [10], C++ [3], Haskell [6] y muchos otros, emplean LLVM como representación intermedia para sus compiladores. Por lo tanto, si es posible verificar un programa en LLVM, también es posible hacerlo para Rust, C++, Haskell, entre otros.

Nota: *Persistimos en la idea de realizar la verificación de ensamblador. Descubrimos que diversas empresas,incluida Intel, llevan a cabo la verificación formal de sus procesadores. Al revisar detenidamente los documentos y presentaciones de estas empresas [5], logramos obtener claridad sobre las posibilidades y metodologías relacionadas con la verificación formal. Este proceso no solo contribuyó a disipar incertidumbres, sino que también proporcionó un enfoque más informado sobre qué se puede lograr y cómo llevar a cabo la verificación formal de manera efectiva.*

2. Detalles Técnicos y versionamiento

2.1. Sistema Operativo y versiones

Se utilizó el sistema operativo "MacOS Sonoma" versión 14.1.1, con la versión del kernel "Darwin23.1.0". Se utilizó la versión de clang "clang-15.0.0" y la versión de llvm "llvm-16.0.0". Se utilizó la versión de OPAM "2.1.5". Se utilizó la versión de Coq "8.15.2". Se utilizó la versión de Ocaml "4.13.1". Se utilizó make "3.81".

3. Instalación de herramientas

Las instrucciones que se encuentran en el repositorio de VeLLVM [11] son muy claras, sin embargo hay que seguirlas en un orden distinto al que se presentan, he aquí el orden para instalar VeLLVM:

1. Instalar "OPAM" [9] (Ocaml Package Manager) con el gestor de paquetes en su sistema operativo, en mi caso "Homebrew" [7] con el comando "brew install opam".
2. Inicializar "OPAM" con el comando "opam init".
3. Agregar el repositorio de coq con el comando "opam repo add coq-released https://coq.inria.fr/opam/released".

4. Crear un Switch para la instalación de VeLLVM con el comando “opam switch create vellvm ocaml-base-compiler.4.13.1”.
5. Clonar el repositorio y los submodulos con el comando “git clone --recurse-submodules git@github.com:kurtcovayne/cm0081-2023-2-lab3.git”.
6. Ingresar al directorio src con el comando “cd cm0081-2023-2-lab3/vellvm/src”.
7. Instalar las dependencias con el comando “make opam”.
8. Para compilar hubieron muchos problemas, pues tal parece que el make no está compilando las cosas en orden, lo que nos sirvió para arreglar esto fue ejecutar varias veces el comando “make -j<n>” donde “n” e ir aumentando el n en caso de que falle, en nuestro caso “n” llegó a ser 24.

4. Verificación de programas en representación intermedia de LLVM

Para verificar un programa en representación intermedia de LLVM se debe seguir los siguientes pasos:

1. Crear un archivo con extensión “.ll” con el código en representación intermedia de LLVM.
2. Compilar el archivo “.ll” con el comando “clang -S -emit-llvm -o <nombre de archivo de salida><nombre de archivo de entrada>”.
3. Se tuvó problemas en “MacOS Sonoma” pues clang agrega modificadores a los parametros cuando ejecuta la instrucción call, también llvm agrega modificadores a los loops y VeLLVM no logra reconocerlos, por lo que se debe arreglar el archivo “.ll” para que no tenga estos modificadores.
4. Agregar comandos de verificación al archivo “.ll”
5. Ejecutar el comando “./vellvm –test-file <nombre de archivo de entrada>”.

5. Programas de interés

Se verificaron los siguientes programas:

1. Programa que realiza potenciación binaria de un número.
2. Programa que calcula el gcd de dos números usando el algoritmo extendido de Euclides.
3. Programa que calcula si existe una solución entera para la ecuación diofántica $ax + by = c$.
4. Programa que calcula la multiplicación de dos números usando la transformada rápida de Fourier en un algoritmo conocido como “Schönhage-Strassen algorithm”.

Los programas en C se pueden encontrar en este repositorio

6. Programas de futuro interes

Será particularmente interesante abordar problemas de ecuaciones diferenciales, donde se pueda verificar la solución para un espacio de entradas dado. Además, se buscará extender la aplicación de la verificación formal a programas con precondiciones sobre un grafo, permitiendo la formulación y verificación de algoritmos que operan en este tipo de estructuras.

Algunos ejemplos concretos de programas futuros incluyen:

1. Desarrollo de un programa que calcule la solución de una ecuación diferencial de primer orden, abarcando cualquier entrada dentro de un intervalo determinado.

2. Implementación de un programa que encuentre el mínimo cubrimiento en un grafo, estableciendo precondiciones como la condición de ser un árbol o ser k-coloreable. Se buscará demostrar formalmente la terminación y complejidad del algoritmo a nivel de implementación

7. Verificación de programas en Brainf**k

En perspectiva, también se contempla la implementación de un programa de verificación formal en el lenguaje Brainf**k, un lenguaje de programación minimalista que se caracteriza por ser Turing completo. Este enfoque abrirá la posibilidad de explorar la verificación formal en un contexto más desafiante y peculiar, aprovechando las capacidades de este lenguaje para expresar algoritmos de manera concisa.

La inclusión de programas en Brainf**k ampliará el alcance de la verificación formal, ofreciendo una perspectiva única y desafiante en la validación rigurosa de algoritmos en lenguajes de programación menos convencionales.

Referencias

[1] *5. C to assembly*. URL: https://www.youtube.com/watch?v=wt7a5BOztuM&ab_channel=MITOpenCourseWare.

[2] «BrainCoqulus: A Formally Verified Compiler of Untyped Lambda Calculus to Brainfuck». En: (). URL: https://read.seas.harvard.edu/~kohler/class/cs260r-17/projects/braincoqulus.pdf.

[3] *C++*. URL: https://cplusplus.com/.

[4] *Clang Call*. URL: https://clang.llvm.org/docs/ClangCommandLineReference.html.

[5] *Formal Verification of x86 Machine-Code Programs*. URL: https://www.cs.utexas.edu/~byoung/cs429/shilpi-slides.pdf.

[6] *Haskell*. URL: https://www.haskell.org/.

[7] *Homebrew*. URL: https://brew.sh/.

[8] *LLVM Loop*. URL: https://llvm.org/doxygen/classllvm_1_1Loop.html.

[9] *Opam*. URL: https://opam.ocaml.org/.

[10] *Rust*. URL: https://www.rust-lang.org/.

[11] *The Vellvm (Verified LLVM) coq development*. URL: https://github.com/vellvm/vellvm.