Compiladores
Proyecto 02

Semestre 2026-1

Verificación de compiladores

¿Qué estrategias se utilizan para demostrar que un compilador es correcto, es decir, que preserva el significado del programa fuente?

Universidad Nacional Autónoma de México
Facultad de Ciencias

Profesor
Manuel Soto Romero

Ayudantes

- Diego Méndez Medina
- Jose Manuel Evangelista Tiburcio
- José Alejandro Pérez Márquez
- Fausto David Hernández Jasso

Integrantes

- Ana Sofía Hernández Zavala — No. de cuenta: 319316717
- Daniela Alejandra Sanluis Castillo — No. de cuenta: 320091179
- Víctor Manuel Mendiola Montes — No. de cuenta: 320197350

Índice

1. Introducción

Los compiladores son un componente crucial para el desarrollo de software, pues se encargan de traducir nuestros lenguajes de programación de alto nivel a código que la máquina sea capaz de ejecutar. Dado el trabajo tan importante que tienen, es bastante complejo el desarrollo de uno de estos sistemas, lo que los vuelve susceptibles a errores que pueden se pueden traducir en comportamientos incorrectos en programas que originalmente funcionan correctamente. Este fenómeno es conocido como *miscompilation* [erts2016]

La precisión de un compilador adquiere una gran importancia para el desarrollo en general, pues el código generado debe preservar el significado o comportamiento del programa fuente para evitar *miscompilation*. Aunque puede parecer relativamente trivial, la importancia de este atributo no puede subestimarse. Como señalan Aho, Lam, Sethi y Ullman [compilers] : *"Es trivial escribir un compilador que genere código rápido si el código generado no tiene por qué ser correcto."* Lograr una precisión perfecta en los compiladores es sumamente complicado, a tal grado que es casi imposible que ningún compilador este libre de errores.

En sistemas de suma importancia para la sociedad (como sistemas de seguridad, sistemas de datos bancarios, etc.), las implicaciones de un *miscompilation* son sumamente graves. Por ello, estándares de seguridad funcional como DO-178C (industria aeroespacial), ISO 26262 (industria eléctrica en los automóviles) o IEC 61508 (industria eléctrica) piden justificar que la traducción realizada por un compilador no introduzca riesgos que no puedan ser controlados.

En este trabajo vamos a examinar estrategias fundamentales para demostrar que un compilador preserva el significado de un programa fuente. Realizamos un introducción de conceptos teóricos necesarios, detallamos el concepto de que un compilador sea correcto, y realizamos un análisis sobre en torno a cuatro enfoques principales para rectificar que un compilador sea correcto. Dichos enfoques son: pruebas formales clásicas, verificación formal a escala industrial, exactitud de implementaciones dirigidas por sintaxis, y técnicas de validación complementarias. Finalmente, discutimos las fortalezas y limitaciones de estos enfoques, junto con nuestras conclusiones.

2. Marco Teórico

Existen compiladores de alto y bajo nivel. El nivel describe qué tan abstracto es el lenguaje fuente con respecto al hardware de la computadora; Es decir, los compiladores que procesan lenguajes de alto nivel están diseñados para ser más legibles para las personas, y un compiladores que procesan lenguajes de bajo nivel tienen un enfoque más directo y detallado sobre el hardware de la computadora.

Para realizar poder verificar la fidelidad de un compilador, requerimos de teoría de lenguajes de programación, lógica matemática e ingeniería de software. Iniciemos rectificando las definiciones de verificación y validación de un compilador.

En el trabajo Leroy [manual], se explica que la verificación de un compilador es probar, con certeza matemática y asistido por máquina, que el código ejecutable producido por el compilador se comporta exactamente como lo especifica la semántica del programa fuente. La validación de un compiladore se describe como una *"validación a posteriori"* del ejecutable producido, pues dado que fases como el ensamblado y el enlazado se realizan con herramientas externas, tenemos que validar el ejecutable resultante y proporcionar seguridad adicional. La verificación y la validación son demasiado importantes en el desarrollo de compiladores para garantizar la correctitud, el rendimiento y la seguridad.

2.1. Estructura de un compilador

En el libro de Aho, Lam, Sethi y Ullman [compilers], el proceso de compilación se puede explicar, de manera muy general, como una secuencia de 6 fases:

- `Análisis Léxico:` Transforma una secuencia de caracteres sin procesar, que es nuestro código de alto nivel, en una secuencia de tokens.
- `Análisis Sintáctico:` Recibe la secuencia de tokens para verificar si se puede generar con la gramática del lenguaje. Para esto, construye una representación estructural que suele ser un árbol sintáctico donde se puede ver claramente la jerarquía y asociación con los constructos del programa.
- `Análisis Semántico:` Se verifica la consistencia del árbol sintáctico con las reglas del lenguage, como la comprobación de tipos, coherencia en los operadores, parámetros, etc. Si existiera algún error, los reporta.
- `Generación de Código Intermedio:` Se construyen una o más representaciones intermedia del programa (en bajo nivel), que podemos considerar como un programa para una máquina abstracta.

- `Optimización del código:` Se transforma la representación intermedia del programa en un programa equivalente, pero que resulta ser más eficiente (generalmente en tiempo de ejecución).
- `Generación de Código:` La representación optimizada se traduce al lenguaje objetivo. Si se traduce a lenguaje máquina, asignamos direcciones de memoria, registros, etc.

2.2. Preservación semántica

La noción semántica es central en este tema, pues nos proporciona una definición del comportamiento de los programas. Para poder saber si un compilador es correcto, requerimos comparar las semánticas del programa fuente y del programa traducido. Otro concepto de suma importancia son los modelos matemáticos, pues nos sirven para describir las unidades léxicas en los programas y los algoritmos que utiliza el compilador para poder reconocer dichas unidades.

Una formulación bastante importante en el tema central de nuestro trabajo es el teorema de preservación semántica (*semantic preservation*). Según los trabajos [Erts 2018], [Erts 2016] y [manual], la definición formal de dicho teorema es *"Si el compilador produce código compilado C a partir del código fuente S, sin informar errores en tiempo de compilación, entonces todo comportamiento observable de C es idéntico a un comportamiento permitido de S, o mejora dicho comportamiento permitido de S al reemplazar comportamientos indefinidos por comportamientos más definidos."*

Esta definición implica que si hemos verificado formalmente que el programa fuente S cumple una propiedad (por ejemplo, no dividir ningún valor entre 0), este teorema garantiza que el código ejecutable C también cumple esa propiedad.

2.3. El trabajo de McCarthy y Painter

En 1967, John McCarthy y James Painter publicaron un artículo sobre la exactitud de un compilador para expresiones aritméticas [mcpain] . Según los trabajos [erts2016], [2676] y [manual], este trabajo fue el primero en establecer el paradigma de comparar la ejecución de la semántica fuente con la ejecución de la semántica destino.

McCarthy y Painter definieron una sintaxis abstracta para las expresiones del lenguaje fuente, utilizando los siguientes predicados: $isconst(e)$ para saber si e es constante; $isvar(e)$ para saber su e es variable; y $issum(e)$ para saber si e es suma. También formalizaron el estado de la máquina mediante un vector de estado, defi-

niendo funciones de acceso y modificación de registros.

El teorema central de su trabajo, que explicamos a más detalle en la sección 3, postula una condición de consistencia. Si ξ es el vector de estado del lenguaje fuente y η es el vector de estado de la máquina, y ambos son consistentes, entonces la ejecución del código compilado transforma el estado de la máquina de tal manera que el acumulador contiene el valor de la expresión fuente. Matemáticamente, su teorema se expresa como:

$$outcome(compile(e,t),\eta) \ =_t \ a(ac, value(e,\xi),\eta)$$

2.4. Enfoque algebraico

Los investigadores Chirica y Martin [5397] propusieron el uso de gramáticas de atributos definidas en un marco de orden algebraico para especificar compiladores de manera modular. En su trabajo, distinguieron entre la exactitud de la especificación del compilador y la exactitud de la implementación.

Dado que un compilador opera mediante una secuencia de activaciones de rutinas semánticas durante el análisis sintáctico, la prueba de su exactitud se convierte en una prueba de corrección parcial de este programa mediante aserciones inductivas. Ellos postulan que con invariantes de traducción se asegura que en cada paso del análisis, el estado interno del compilador mantiene una relación coherente con el fragmento de programa fuente procesado hasta el momento. Con este enfoque, se permitió escalar las pruebas de la exactitud de un compilador más allá de expresiones simples hacia construcciones de lenguaje más complejas.

3. Correctitud de los Compiladores

Como mencionamos anteriormente, el trabajo de McCarthy y Painter [mcpain] fue el primero en introducir el paradigma sobre que tan correcto es un compilador. En la sección 2, vimos que en su trabajo definen (y demuestran) un teorema para rectificar la exactitud de un compilador para expresiones aritméticas, que vamos a desglosar más detalladamente.

Textualmente, el teorema dice lo siguiente: *"Si η y ξ son vectores de estado de la máquina y del lenguaje fuente, respectivamente, tales que*

$$c(loc(v,\eta) \ = \ c(v,\xi)$$

entonces

$$outcome(compile(e,t),\eta) \ =_t \ a(ac, value(e,\xi),\eta)$$

En este [mcpain] y más versiones del mismo articulo la formula de la hipótesis viene incompleta, pero dado el contexto, asumimos que es $c(loc(v,\eta),\eta) = c(v,\xi)$. Una vez aclarado esto, vamos a detallar el teorema para que sea un poco más digerible.

Primero, se mencionan dos estados: ξ es el estado del lenguaje fuente y η es el estado de la memoria de la máquina. Lo siguiente en la hipótesis es la condición

$$c(loc(v,\eta),\eta) = c(v,\xi)$$

donde v es una variable en el lenguaje fuente; $loc(v,\eta)$ es la ubicación en η donde está guardada la variable v; $c(v,\xi)$ es el valor de la variable v en el estado ξ (analogo para $c(loc(v,\eta),\eta)$) . En términos más simples, postula que para toda variable v, lo que valga en el lenguaje fuente debe coincidir con lo que hay en la máquina en el lugar donde v está almacenada.

Ahora, pasando a la parte que afirma el teorema, tenemos

$$outcome(compile(e,t),\eta) =_t a(ac, value(e,\xi),\eta)$$

donde $compile(e,t)$ es la función que genera el código máquina para la expresión e, utilizando registros temporales a partir de la dirección t; $outcome(compile(e,t),\eta)$ es el estado final de la máquina después de ejecutar el código compilado, comenzando desde el estado η; $=_t$ significa que los dos estados de la máquina son idénticos en todos los registros, excepto los que son destinados a los temporales; $value(e,\xi)$ es el cálculo del valor de la expresión e según la semántica del lenguaje fuente; $c(ac, value(e,\xi),\eta)$ es la función que regresa el estado resultante al tomar el estado η y cambiar el contenido de un acumulador ac por el valor calculado. En palabras más sencillas, quiere decir que al ejecutar el código compilado para e deja en el acumulador el mismo valor que la evaluación fuente de e.

Este teorema es una definición muy formal y elegante sobre la exactitud de un compilador por varias razones, pero la principal es que demuestra que hay una equivalencia entre la sintaxis ejecutada y la semántica abstracta, garantizando que el compilador no altera el significado del programa original. Esto es precisamente lo que dice el teorema de preservación semántica. Entonces en términos generales, la exactitud de un compilador se refiere a la propiedad de preservación semántica a lo largo de la traducción del código fuente. En las siguiente sección, examinaremos diferentes estrategias utilizadas para conseguir esta propiedad.

4.

5. Diseño e Implementación del Código

En la carpeta `/Proyecto-02-Investigacion-Academica/src` se tienen 2 archivos: `MiniCompilador.hs` y `compilador.hs`.
MiniCompilador.hs es un compilador que traduce las expresiones del programa a instrucciones con la finalidad de ver su correctitud al comparar la ejecución directa (AST) con la ejecución compilada (Bytecode).

compilador.hs es un compilador muy pequeño que nos sirvió de ejemplo para hacer el compilador más grande, éste solo maneja sumas, sin embargo se implementaron todos los pasos de un compilador.

5.1. Código del programa MiniCompilador.hs

5.1.1. Estructura

1. Lenguaje fuente (AST): Define la estructura de cómo deben verse los programas en el lenguaje; no se ejecuta nada, solo se define la forma del lenguaje.

2. Intérprete del AST: Ejecuta el lenguaje fuente, sin haberlo compilado; toma el programa, ejecuta cada línea del mismo una por una, mantiene el entorno de variables y devuelve los valores finales de dichas variables.

3. Bytecode y máquina virtual: Ahora se muestra un lenguaje de bajo nivel con su programa correspondiente que lo ejecuta; ejecuta cada instrucción usando una pila y un entorno.

4. Compilador (AST a Bytecode): Convierte los programas AST en instrucciones simples, es decir que traduce las expresiones del programa a instrucciones.

5. Verificación de correctitud: Prueba la correctitud del compilador, comparando la ejecución directa (AST) con la ejecución compilada (Bytecode); si resultan iguales, entonces el compilador es correcto.

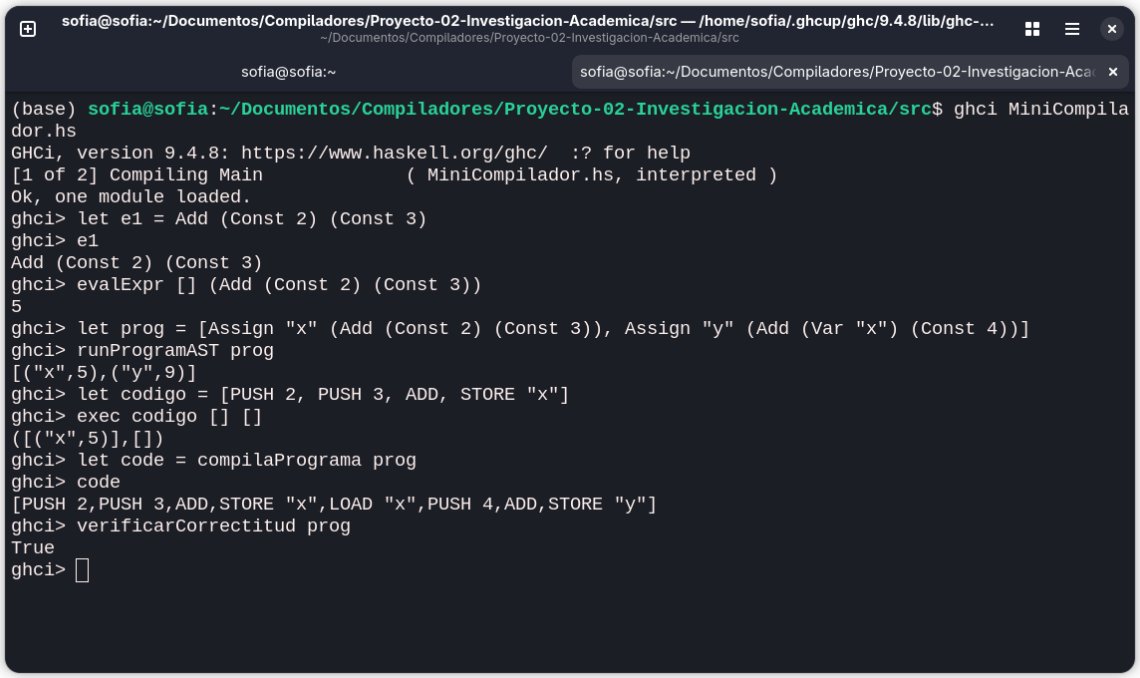

Figura 1: Ejecución con ejemplos de MiniCompilador.hs

5.1.2. Código del programa compilador.hs

5.2. Verificación del Código

6. Conclusiones