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* [1]

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 [2]: *"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 [3], 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 [2], 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 [3], 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. Enfoque algebraico

Los investigadores Chirica y Martin [4]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

En 1967, John McCarthy y James Painter publicaron un artículo sobre la exactitud de un compilador para expresiones aritméticas [5]. Según los trabajos [1], [6]y [3], este trabajo fue el primero en establecer el paradigma de comparar la ejecución de la semántica fuente con la ejecución de la semántica destino.

En este 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:

*"**Teorema 1.** 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 [5]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 el teorema es la hipótesis con la siguiente 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. Estrategias para que un compilador sea correcto

4.1. Inducción matemática

La primera prueba de exactitud en un compilador fue hecha para uno que analizaba expresiones aritméticas, y fue realizada solamente con lógica matemática. Para poder hacer esto, se tiene que definir formalmente la semántica del lenguaje fuente y la del lenguaje objeto. Luego se especifica el algoritmo de compilación y se demuestra un teorema de correspondencia entre ambos niveles. McCarthy y Painter [5]hacen esto en su trabajo de 1967, pero la estrategia seguida fue usar una sintaxis abstracta en lugar de una sintaxis concreta, que se define mediante predicados y funciones asociadas.

En concreto, definen los predicados: $isconst(e)$, para saber si e es constante; $isvar(e)$, para saber si e es variable; y $issum(e)$, para saber si e es suma. La única función asociada es para una suma: $s1(e)$ y $s2(e)$ para extraer los sumandos. Para el lenguaje objeto (que tiene un acumulador y las instrucciones *li*, *load*, *sto* y *addse*) trabaja con una sintaxis analítica y sintaxis sintética. La sintaxis analítica nos permite desarmar una instrucción y la sintaxis sintética nos da funciones constructoras. A todo lo anterior, se añaden las definiciones de los estados y funciones que vimos en la sección 3, para poder definir el teorema descrito en la misma sección.

La prueba para demostrar que este compilador es correcto, se aplica inducción sobre la sintaxis abstracta.

- **Casos bases:** e es una constante o variable.($isconst(e)$ ó $isvar(e)$). Es trivial ver que el código generado realiza correctamente la carga del valor.

- **Paso inductivo:** e es una suma ($issum(e)$), se asume como hipótesis de inducción que el compilador es correcto para los sumandos $s1(e)$ y $s2(e)$. Luego, se expande la ejecución del código generado para la suma, que implica almacenar resultados y sumar, para utilizar transitividad de las transformaciones de estado para demostrar que el resultado final cumple produce el resultado correcto en el acumulador ac.

Al finalizar, se concluye que para cualquier expresión válidavel compilador entregará código cuya ejecución computa el mismo valor que la evaluación directa de la expresión.

Estas pruebas hechas a mano demostraron una vialidad conceptual en la exactitud de los compiladores, pero están limitadas a lenguajes relativamente simples y compiladores no propiamente programados. A medida que los lenguajes y compiladores se volvieron más complejos, estas pruebas ya no eran una muy buena opción por la cantidad de pasos a considerar. Sin embargo, estas primeras pruebas fueron la base teórica e inspiración para la verificación con métodos más avanzados.

4.2. Lógica mecanizada

En los años 70s, Los autores Milner y Weyhrauch publican un trabajo [7]donde introducen la mecanización de las pruebas utilizando lógica para funciones computables (*Logic for Computable Functions* o LCF). Esta lógica esta basada en el cálculo lambda tipado y una regla de inducción de Dana Scott, y esta diseñada para razonar sobre funciones computables parciales y recursivas.

El lenguaje fuente S tiene asignaciones, condicionales, ciclo while y composiciones de declaraciones. Un programa bien formado de S sería cualquiera que este construido con esos constructores, y además sus expresiones son nombres o aplicaciones de operadores binarios a subexpresiones. Semánticamente, el significado de un programa se modela como una función de estado (*statefunction*), que va de estados de entrada a estados de salida. Un estado sv es una función del conjunto de nombres al conjunto de valores. La semántica de expresiones se da con $MSE(e, sv)$: evalúa la expresión e en el estado sv. La semántica de las expresiones esta definida con $MSE(e, sv)$, donde se evalúa la expresión e en el estado sv. La semántica de los programas esta dada por $MS(p)$, donde para cada para cada declaración se define la función de estado correspondiente.

El lenguaje objetivo T es un ensamblador simple con saltos y una pila. Se define un *store* como un par $(sv|pd)$, donde sv es un estado y pd una pila. Semánticamente, el significado de un programa de T es una función store (*storefunction*), que va de *stores* a *stores*. En el trabajo se plantean las instrucciones JF, J, $FETCH$, $STORE$, $LABEL$ y DO con su efecto sobre (sv, pd), ý se define formalmente $MT(p)$

como la semántica de un programa p en T.
Es útil señalar que en lugar de utilizar conjuntos complejos de funciones mutuamente recursivas, como sugería McCarthy para los diagramas de flujo, mejor se define la semántica del programa objeto MT mediante una única definición recursiva. Utilizan una función auxiliar $find$ que busca la etiqueta de destino en la lista de instrucciones, lo que permite modelar la semántica de un lenguaje con flujo de control arbitrario para la lógica de las funciones recursivas de LCF.

Para definir la exactitud de un compilador, lo que hicieron fue declarar algoritmos de compilación *comp* y *compe* (ambos de S a T), dicen que el compilador es correcto si al ejecutar el programa fuente y el programa compilado se produce el mismo estado final, iniciando la máquina destino con una pila vacía. Formalmente:

$$(MS(p))(sv) \equiv svof((MT(comp(p)))(sv|NIL))$$

Lo anterior dice que el resultado de correr p en S desde sv tiene que ser equivalente con la componente de estado del *store* resultante de correr $comp(p)$ en T desde el *store* inicial $(sv|NIL)$. Con esta idea en mente, introducen una función $SIMUL$ donde dada una función *store* g, la convierte en una función de estado:

$$SIMUL(g)(sv) = svof(g(sv|NIL))$$

Con esto, el que un compilador sea correcto se reescribe como:

$$MS = comp \otimes MT \otimes SIMUL$$

reduciéndose todo a demostrar que *comp*, MT y $SIMUL$ son homomorfismos.

Esta estrategia demuestra que las pruebas de compiladores, que son propensas a errores cuando se hacen a mano, pueden ser verificadas por una computadora.

4.3. Especificación e implementación de un compilador dirigido por sintaxis

En 1986, Chirica y Martin [4]proponen una estrategia que aborda la especificación del compilador y su implementación concreta en un lenguaje de programación. Esta estrategia se centra en compiladores dirigidos por sintaxis, implementados mediante LR(k) o LL(k).

En términos muy resumidos, la especificación del compilador $c-spc$ se formaliza utilizando una definición con gramáticas de atributos. El compilador se ve como una función que va del árbol de derivación sintáctica del programa fuente a un programa objeto. Es decir, que para cada programa fuente se "dicta" cuál debe ser el

programa objeto correspondiente. Por lo general, la exactitud de un compilador lo refieren con la exactitud de la especificación.

La implementación de un compilador $c-imp$, que es el enfoque de su trabajo, se modela como una secuencia de activaciones de rutinas de traducción (*Translation Routines* - TRs). En LR, estas rutinas se ejecutan en las reducciones. Las rutinas operan sobre un entorno de traducción (*Translation Environment* - TE), que es una colección de estructuras de datos globales. La ejecución del compilador sobre un programa fuente nos lleva a una secuencia lineal de llamadas a estas rutinas, transformando el entorno de traducción inicial en uno final que contiene el código objeto.

La exactitud del compilador se descompone entonces en verificar dos cosas: La especificación del compilador realmente preserva la semántica y la implementación del compilador coincide con la especificación. Formalmente:

$$c - gen_{prog}(c - imp_{prog}(t)) \;=\; c - spc_{prog}(t)$$

donde t es un programa fuente y $c-gen$ extrae el comportamiento de la implementación el programa objeto a comparar.

Para poder demostrar esa exactitud de implementación, es necesaria la formulación de invariantes de traducción $Q_A(\gamma)$ (*translation invariants*), que describen para cada símbolo no terminal A de la gramática, se define un predicado $Q_A(\gamma)(u_0, u)$ que relaciona el estado del entorno de traducción antes (u_0) y después (u) de procesar una sub-estructura de tipo A, asegurando que el código generado en el entorno corresponde al código γ definido por la especificación abstracta. En el trabajo se define un teorema principal, que parafraseándolo dice algo como: $c - imp$ es una implementación correcta de $c - spc$ si para cada no terminal A existe un invariante Q_A, tal que para cada producción se cumple una obligación de exactitud parcial.

Usando técnicas de verificación de programas convencionales, se demuestra que bajo estas invariantes, el programa que representa la implementación del compilador es parcialmente correcto con respecto a la especificación. A medida que se reconoce una producción de la gramática y la rutina de traducción asociada, se verifica que el estado de traducción resultante cumple el invariante que corresponde a haber traducido esa construcción correctamente.

La ventaja de este enfoque es que si el lenguaje fuente está bien definida por una buena gramática y reglas de traducción, la prueba de exactitud puede dividirse por producciones gramaticales. Además, dado que muchos compiladores son esencialmente traducciones dirigidas por sintaxis, esta metodología es bastante general para compiladores de lenguajes tradicionales. Sin embargo, una limitación de esta

estrategia es que asume que la especificación del compilador ya es correcta respecto a la semántica.

4.4. CompCert

En los últimos años, los avances tecnológicos permitieron abordar la verificación de compiladores industriales completos. Un "santo grial" es CompCert, un compilador para el lenguaje C, escrito principalmente en Coq y extraído a OCaml para su ejecución, que fue desarrollado por Xavier Leroy y colaboradores [1,3,8]. Este compilador es reconocido como el primer compilador optimizador formalmente verificado. Dicha prueba demuestra que el código ejecutable generado es semánticamente equivalente al programa fuente en C, evitando cualquier posibilidad de *miscompilation*.

El enfoque de CompCert se basa en definir rigurosamente la semántica de todos los lenguajes intermedios involucrados en sus múltiples fases de compilación. En total, cuenta con 20 fases de compilación que conectan los archivos fuente de C con el código objeto pasando por 11 lenguajes intermedios. Las fases se pueden agrupar en 4 etapas sucesivas:

- Análisis sintáctico. Se generan tokens, construye un AST, y realiza el chequeo de tipos.
- Compilador front-end de C. Se vuelven a revisar tipos, se fija un orden de evaluación permitido por C y hace conversiones y comportamientos dependientes de tipos para producir un código intermedio *Cminor*.
- Compilador back-end. Compuesto de 12 fases, donde se realizan optimizaciones y transforma progresivamente el control y los datos hasta llegar a un AST de ensamblador.
- Ensamblaje. Imprime el ensamblador en sintaxis concreta, añade información de depuración y encomienda a un ensamblador y enlazador externos.

Para cada una de las 20 fases en estos 4 grupos, se prueba el teorema de preservación semántica, pues al probar cada fase del compilador individualmente, se logra dividir el problema en pequeños problemas que son más manejables. Concretemente, se construye un diagrama de simulación para cada fase, donde se analiza las relaciones de simulación entre los estados del programa antes y después de la transformación, con el fin de garantizar que ninguna fase introduzca un comportamiento indebido. Al componer en secuencia todas estas simulaciones individuales se obtiene la preservación semántica global del compilador completo.

Dada la complejidad de un compilador real, una prueba formal es un esfuerzo gigantesco. El desarrollo de CompCert requirió de 6 años de trabajo y aproximadamente 100,000 líneas de scripts de prueba en Coq . Un resultado importante fue

demostrar que pruebas de este tamaño no son viables realizarlas a mano, pues se necesita la rigurosidad mecanizada de un asistente de pruebas para evitar errores y manejar la complejidad. Como parte del proyecto, se formalizó una semántica operacional para un gran subconjunto de C, denominado CompCert C, y para los lenguajes máquina. Esta es una contribución significativa, ya que equivale a un fragmento formal del estándar de C y de las especificaciones de CPU, contra las cuales se puede razonar.

Al formalizar el compilador en Coq, todas las propiedades y casos se checan exhaustivamente. Leroy señala que esta mecanización aporta una confianza casi absoluta en la validez de la prueba, pues se eliminan errores humanos en el razonamiento. De hecho, citando el trabajo [3]postulan lo siguiente: "A principios de 2011, la versión en desarrollo de CompCert es el único compilador que hemos probado para el cual Csmith (un generador de casos de prueba) no puede encontrar errores de código incorrecto.". Esto, por ejemplo, elimina la necesidad de ciertas actividades de validación a nivel de código objeto.

Aun con estas garantías, es importante reconocer las limitaciones actuales de CompCert. En la versión actual, alrededor del 90 % de las rutinas del compilador han sido demostradas correctas, incluyendo todos los algoritmos de optimización y generación de código, pero quedan partes sin verificar. Específicamente, ciertas transformaciones de alto nivel previas a la fase verificada y las fases finales de ensamblado y enlazado no están cubiertas por la prueba formal. Algunas de estas partes quedan fuera por falta de especificaciones o por ser añadidos recientes al compilador.

En consecuencia, CompCert combina su núcleo probado con técnicas convencionales para las partes no verificadas. Por ejemplo, dado que algunas partes del proceso son difíciles de verificar formalmente dentro de Coq, como el ensamblado y enlazado final, se utiliza la validación de la traducción mediante herramientas como Valex. Su trabajo, es analizar código ensamblador abstracto generado por CompCert y el binario enlazado final, verificando a posteriori que no se han introducido discrepancias. Esto proporciona una capa adicional de seguridad para las fases que escapan al núcleo verificado formalmente.

A pesar de ser incompleto, la verificación de CompCert demuestra mejoras de exactitud enormes comparado con compiladores ordinarios, lo que resulta valioso en ambientes donde la exactitud es crítica. Las técnicas y experiencias de este proyecto sientan las bases para que en el futuro más compiladores sean objeto de verificación formal.

4.5. Compositional CompCert

Un desafió para CompCert es la compilación separada, pues en sus pruebas asumen que todo el programa se compila a la vez. Sin embargo, en el mundo real los programas se construyen enlazando módulos compilados independientemente, a menudo escritos en diferentes lenguajes. Con esto en mente, Stewart, Beringer, Cuellar y Appel y col. en 2015 [6]desarrollaron una variante de CompCert que aborda la exactitud de un compilador en escenarios de compilación separada y módulos enlazados.

La estrategia para abordar esto se basa en la semántica de interacción, pues este modelo define el comportamiento de un módulo no de forma aislada, sino en términos de su protocolo de comunicación con el entorno, como llamadas a funciones externas o acceso a memoria compartida. Se introduce un operador de enlace independiente del lenguaje que define cómo se componen estos módulos semánticos. Esto permite demostrar que si cada módulo es compilado correctamente, entonces el programa completo resultante al enlazarlos también es correcto.

Para probar la exactitud en este contexto, no basta trabajar con relaciones de simulación simples. Se tienen que introducir simulaciones estructuradas que refinan las relaciones de simulación lógica incorporando una disciplina de Rely-Guarantee. Dicha disciplina formaliza la relación entre un módulo y su entorno, pues el módulo garantiza ciertas propiedades siempre y cuando el entorno satisfaga ciertas suposiciones. También se enriquecen las simulaciones con datos de propiedad para gestionar regiones de memoria privada introducidas por el compilador, asegurando que el enlazado no viole la integridad de estas estructuras internas.

Todos los resultados de Compositional CompCert fueron formalizados en Coq, mostrando que es viable escalar la verificación formal a arquitecturas de software más grandes y de desarrollo independiente. Aunque estos avances aún academicos, inician el camino para compiladores verificados más flexibles, capaces de integrarse en entornos de software complejos típicos de la industria.

5. Desarrollo del ejemplo práctico

En la carpeta `/Proyecto-02-Investigacion-Academica/src` definimos 2 archivos: `CompiladorElaborado.hs` y `CompiladorSimple.hs`. Las instrucciones para ejecutarlos esta estan en el archivo README del proyecto.

En `CompiladorSimple.hs` implementamos casi exactamente el mismo paradigma que en [5], pues comparamos el resultado de la semántica fuente con el resultado de la semántica de la máquina de pila para las expresiones que solo se suman entre si.

Por ejemplo, si corrieramos el programa con la expresión $1 + 2 + 3$, obtenemos lo siguiente:

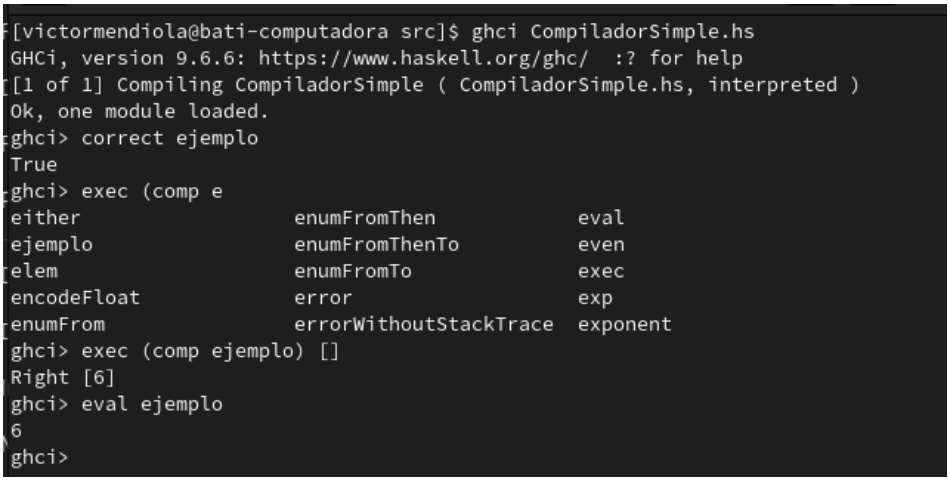

Figura 1: Ejecución de `CompiladorSimple.hs`

- `correct ejemplo`. Esto nos regresa el booleano que determina si el resultado en la pila tras ejecutar el código compilado coincide con la evaluación directa.
- `exec (comp ejemplo) []`. Esto nos dice si el programa compilado deja un único resultado en la pila.
- `eval ejemplo`: Evaluación directa de AST.

Durante el desarrollo de ambos modulos nos dimos cuenta que no manejabamos errores, entonces optamos por usar `Either` para capturar errores por una variable no definida o una pila indefnida.

En `CompiladorElaborado.hs` manejamos un entorno *Env* mediante un mapa *Map*. Esto nos permite demostrar que el compilador preserva los valores aislados y el estado de la memoria tras una secuencia de asignaciones. La función *verificarCorrectitud* es nuestra implementación del teorema de preservación semántica, y devuelve *True* solo si ambos entornos finales son iguales. Por ejemplo, si corremos el programa con `x = 2, y = x + 3`, obtenemos lo siguiente:

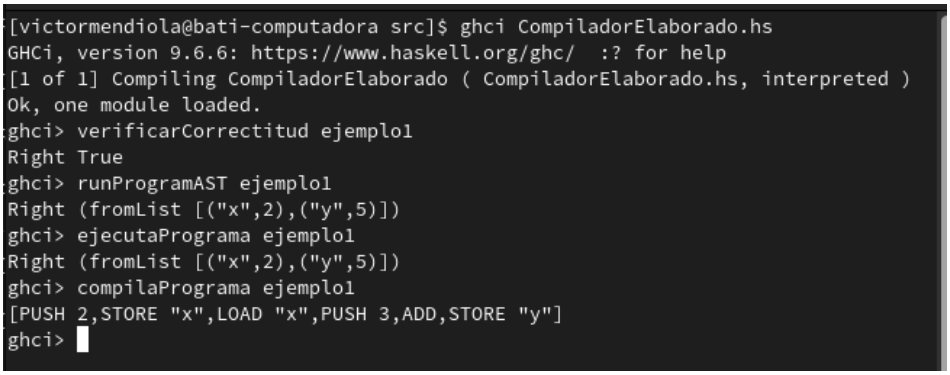

Figura 2: Ejecución de `CompiladorElaborado.hs`

- `verificarCorrectitud ejemplo1` : Esto nos dice si no hubo errores y si los entornos finales son iguales.
- `runProgramAST ejemplo1` : Ejecutamos el programa sin compilarlo con la semántica del lenguaje fuente.
- `ejecutaPrograma ejemplo1` : Ejecutamos el programa compilado
- `compilaPrograma ejemplo1` : Esto muestra qué es lo que traduce el compilador y cómo se ejecuta.

6. Discusiones y limitaciones

Haciendo el análisis a varios enfoques, podemos notar que no existe una única solución para ver la exactitud de un compilador, y mucho menos una estrategia que se pueda considerar perfecta u optima. A pesar de esto, cada estrategia tiene sus fortalezas y debilidades.

Las pruebas a mano nos proporcionan una garantía matemática, pero es muy poco práctica para los compiladores a gran escala. Dichas pruebas solo son viables para lenguajes relativamente pequeños o análisis de casos muy acotados, pues requeriría de un esfuerzo muy grande realizar estas pruebas en los compiladores más apegados a nuestra vida cotidiana, además de ser propenso a errores humanos. A pesar de su gran limitante, esta estrategia estableció buenos fundamentos teóricos que aseguran que se puede demostrar que tan correcto es un compilador. Pasando al otro extremo, la estrategia de CompCert tiene la gran ventaja de ofrecer confianza casi absoluta. La principal desventaja sería su alto costo y mantenimiento, pues el hacer una modificación en el lenguaje que analizará el compilador requeriría de reformular varias pruebas. Aun así, pueden existir partes fuera del compilador como herramientas externas, bibliotecas, etc.

Otra tema a considerar, es que el mismo comportamiento depende de la semántica elegida, pues en lenguajes que podrían tener comportamientos indefinidos se deberían definir cierto límites para considerar que comportamientos tomar como válidos y cuales no.

Nuestra última limitante sería nuestro programa en haskell, pues nuestros "compiladores" son bastante pequeños, pues solo cubrimos un subconjunto de expresiones y asignaciones sin considerar lenguajes más reales (o complejos), optimizaciones, controles de flujo, manejos de memoria, llamadas a funciones, interacciones con el sistema, etc. Si quisieramos considerar todo lo anterior, requeriría de meses de trabajo y más conocimiento de nuestra parte para poder analizar y considerar la mayor cantidad de casos posibles para un solo lenguaje, sin contar la elección del mismo.

7. Conclusiones

En este trabajo abordamos la pregunta central sobre qué estrategias se utilizan para demostrar que un compilador sea correcto. A partir del teorema de preservación semántica, vemos que la exactitud de un compilador no se trara de una simple "intuición", sino de una propiedad que compara de forma muy minuciosa el comportamiento permitido por la semántica del programa fuente con el comportamiento del programa compilado.

Nuestro análisis histórico nos lleva a la conclusión de que existen distintos niveles de garantía con sus respectivos compromisos y alcances: La inducción matemática es bastante útil y formal para lenguajes pequeños, pero que se vuelve bastante frágil con lenguajes más completos; La lógica mecanizada rebasan el límite anterior, pues reducen el error humano y permite sostener casos considerablemente más grandes;

La estrategía con compiladores dirigidos por sintaxis muestran que la exactitud no solo esta dada por la idea de que la traducción sea correcta (especificación), sino que la implementación concreta cumpla esa especificación mediante invariantes de traducción ligados a producciones gramaticales; Finalmente, CompCert muestra que la verificación alcanza escalas industriales al probar la preservación semántica fase por fase y compone dichas pruebas, pues se obtiene una garantía aun más fuerte sobre la presencia de *miscompilation*. A pesar de todo esto, aun se cuenta con algunos casos sin rectificar, haciendo que la perfección absoluta en un compilador sea casi imposible por ahora.

El ejemplo práctico refuerza la idea principal de este trabajo, pues al comparar la evaluación del AST con la ejecución del código compilado podemos entender mejor una idea básica del teorema de preservación semántica. Aunque nuestro objetivo no fue crear un compilador real, podríamos decir que nuestra implementación sí cumple el objetivo de poder comparar la semántica fuente con semántica destino para poder

analizar que tan correctamente esta definido nuestro compilador.

Referencias

[1] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "Compcert – a formally verified optimizing compiler," in *Embedded Real Time Software and Systems (ERTS 2016), 8th European Congress*, Toulouse, France, Jan. 2016, presented at ERTS 2016; P̈resident's favoriteäward. [Online]. Available: https://compcert.org/

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[3] X. Leroy, *The CompCert C Verified Compiler: Documentation and User's Manual*, 2024, hAL Id: hal-01091802. [Online]. Available: https://inria.hal.science/hal-01091802

[4] L. M. Chirica and D. F. Martin, "Toward compiler implementation correctness proofs," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 185–214, Apr. 1986.

[5] J. McCarthy and J. Painter, "Correctness of a compiler for arithmetic expressions," in *Mathematical Aspects of Computer Science*, ser. Proceedings of Symposia in Applied Mathematics. Providence, RI: American Mathematical Society, 1967, vol. 19, pp. 33–41, reprint of a Stanford technical report. [Online]. Available: https://www-formal.stanford.edu/jmc/mcpain.pdf

[6] G. Stewart, L. Beringer, S. Cuéllar, and A. W. Appel, "Compositional compcert," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Mumbai, India: ACM, Jan. 2015, pp. 275–287.

[7] R. Milner and R. Weyhrauch, "Proving compiler correctness in a mechanized logic," in *Machine Intelligence 7*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, 1972, pp. 51–72.

[8] D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy, "Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler," in *ERTS2 2018 – 9th European Congress on Embedded Real-Time Software and Systems*, Toulouse, France, Jan. 2018, pp. 1–9, hAL Id: hal-01643290. [Online]. Available: https://inria.hal.science/hal-01643290