Titulo del trabajo

Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Profesor
Manuel Soto Romero

Ayudantes de laboratorio:
Jason Todd
Dick Grayson

Integrantes
Hernández Zavala Ana Sofía - No. Cuenta: 319316717
Sanluis Castillo Daniela Alejandra - No. Cuenta: 320091179
Mendiola Montes Victor Manuel - No. Cuenta: 320197350

Fecha de entrega : 26/11/25

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

Introducción

Los modelos, la teoría y los algoritmos asociados con un compilador pueden aplicarse a una amplia gama de problemas en el diseño y desarrollo de software. Antes de que se ejecute algún programa, su código debe de ser traducido a alguna forma en la que la computadora pueda ejecutarla; esta traducción de un lenguaje a lenguaje máquina se logra por medio de un *compilador*.
Para demostrar esta pregunta, implementamos un mini compilador en Haskell el cual hace la traducción de un lenguaje simple a bytecode y verifica que la salida del compilador sea correcta, comparando sus ejecuciones.

Fundamentos

Hay compiladores de alto nivel y de bajo nivel, el nivel describe qué tan abstracto es el lenguaje fuente con respecto al hardware de la computadora; es decir que los compiladores que procesan lenguajes de *alto nivel* están diseñados para ser más entendibles por las personas (más cercanos al lenguaje natural) con un alto grado de abstracción. En cambio los compiladores que procesan lenguajes de *bajo nivel* tienen un enfoque más directo y detallado sobre el hardware de la computadora.
Antes de adentrarnos completamnete en la definición de correctitud es necesario saber un poco sobre la verificación.
La *verificación* garantiza que las transformaciones de un compilador sean matemáticamente correctas y sigan las reglas formales, ocupa pruebas lógicas para verificar que el compilador no introduzca errores al traducir el código.
En cambio la *validación* garantiza el correcto funcionamiento del compilador mediante la ejecución de programas de prueba y la comparación de los resultados

esperados y no se centra en la demostración matemática.
La verificación y la validación son demasiado importantes en el desarrollo de compiladores para garantizar la correctitud, el rendimiento y la seguridad.

Correctitud de los Compiladores

De hecho la correctitud es uno de los aspectos importantes de la verificación en compiladores.
La correctitud asegura que el código compilado mantenga la lógica original y el comportamiento esperado del código fuente; es fundamental para la fiabilidad, la seguridad y la integridad del sistema del software. Garantiza que un compilador traduzca el código fuente a código máquina correcto y eficiente, evitando así problemas no deseados.

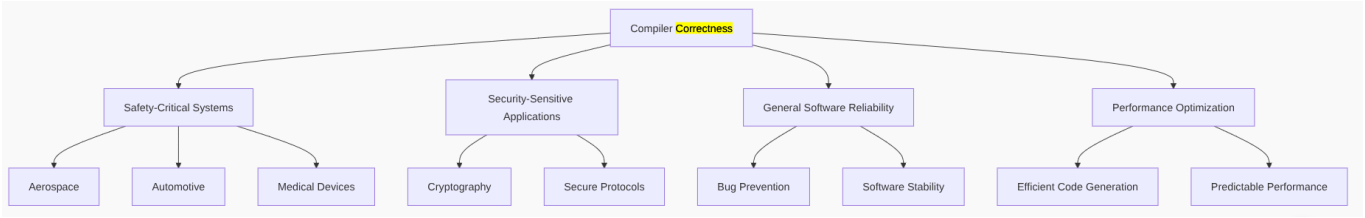

Figura 1: En este diagrama se muestra su importancia

3.1. Pasos para lograr la Correctitud

1. Análisis Sintáctico (Parsing): Verifica que el código sigue las reglas de gramatica del lenguaje.
2. Análisis Semántico: Verifica que el código tenga sentido y operaciones válidas.
3. Verificación de Representación Intermedia (IR): Asegura que las transformaciones mantengan la semántica del programa.
4. Comprobación de equivalencia: Las versiones optimizadas y no optimizadas deben generar resultados idénticos.
5. Gestión de tablas de símbolos: Garantizar un alcance correcto, declaraciones de variables y consistencia de tipos.
6. Validacion del Árbol de Anlálisis Sintáctico (AST): Verifica que la estructura del arbol es correcta.
7. Comprobación de tipos: Garantiza que las operaciones se apliquen a tipos de datos compatibles.

8. Análisis de flujo de control y flujo de datos: Verifica que la ejecución del programa siga las rutas lógicas correctas.
9. Métodos formales: Utiliza pruebas matemáticas para verificar la corrección a nivel teórico.
10. Pruebas con implementaciones de referencia: Compara los resultados con un compilador confiable para validar la precisión.

3.1.1. Metodología estructurada para verificar la correctitud en la compilacion

1. Verificación formal: utiliza pruebas matemáticas para confirmar que el compilador conserva la semántica del programa.
2. Pruebas sistemáticas: ejecuta casos de prueba predefinidos para detectar errores en las distintas etapas de la compilación.
3. Validación de la traducción: compara el comportamiento del programa fuente y el de destino tras la compilación para garantizar su corrección.
4. Pruebas aleatorias (fuzzing): genera entradas aleatorias para descubrir fallos y vulnerabilidades inesperadas del compilador.
5. Herramientas de depuración automatizada: identifica y rastrea los errores introducidos durante la compilación mediante análisis de registros y marcos de depuración.
6. Pruebas diferenciales: ejecuta la misma entrada en varios compiladores y compara las salidas para detectar inconsistencias.
7. Pruebas basadas en propiedades: verifica que el código compilado mantenga las propiedades fundamentales del programa (p. ej., corrección y determinismo).
8. Ejecución simbólica: analiza todas las posibles rutas de ejecución en el código compilado para detectar inconsistencias lógicas.
9. Técnicas de resolución de restricciones: Utiliza solucionadores como Z3 para verificar que las transformaciones del compilador no alteren el comportamiento del programa.
10. Evaluación comparativa (Benchmarking) y Profiling: Evalúa el rendimiento de salida del compilador para garantizar que las optimizaciones sean efectivas y no reduzcan la velocidad de ejecución.

Diseño e Implementación del Código

En la carpeta /Proyecto-02-Investigacion-Academica/src se tiene 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 ejecucion directa (AST) con la ejecucuion 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 si se implementó todos los pasos de un compilador.

4.1. Código del programa MiniCompilador.hs

4.1.1. Estructura

1. Lenguaje fuente (AST): Define la estructura de como deben de 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 linea 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 instruccione simples, es decir que traduce las expresiones del programa a instrucciones.

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

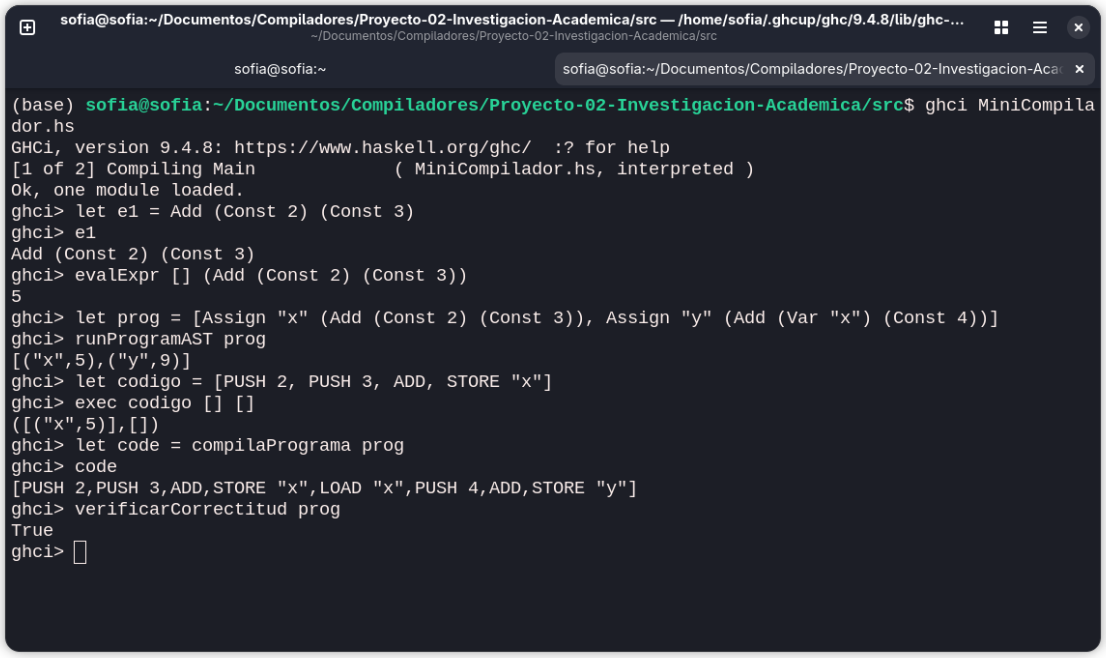

Figura 2: Ejecución con ejemplos de MiniCompilador.hs

4.1.2. Código del programa compilador.hs

4.2. Verificación del Código

Conclusiones