Compilador para el Lenguaje HULK: Implementación con Análisis Léxico, Sintáctico, Semántico y Generación de Código LLVM

Adrián Souto Morales and Gabriel Herrera Carrazana

Universidad de La Habana, Facultad de Matemática y Computación
adrian.souto@estudiantes.matcom.uh.cu,
gabriel.herrera@estudiantes.matcom.uh.cu

Abstract. Se desarrolló un compilador completo para el lenguaje de programación HULK (Havana University Language for Kompilers) utilizando C++ como lenguaje de implementación. El compilador implementa todas las fases tradicionales: análisis léxico mediante un lexer nativo, análisis sintáctico con Bison, análisis semántico con verificación de tipos, y generación de código LLVM IR. Se utilizó el patrón Visitor para la generación de código y se implementó un sistema de tipos robusto que soporta herencia, polimorfismo y verificación estática de tipos. El proyecto se encuentra disponible en GitHub y genera código ejecutable a través de la infraestructura LLVM.

Keywords: Compiladores · LLVM · Análisis Semántico · Generación de Código · Lenguajes de Programación

1 Introducción

El desarrollo de compiladores constituye una de las áreas fundamentales de las ciencias de la computación, combinando teoría de lenguajes formales, algoritmos de análisis y técnicas de optimización. En este trabajo se presenta la implementación de un compilador completo para HULK, un lenguaje de programación diseñado con fines educativos que incorpora características modernas como inferencia de tipos, programación funcional y orientada a objetos.

El objetivo principal fue desarrollar un compilador que no solo procesara correctamente la sintaxis del lenguaje, sino que también implementara un sistema robusto de verificación semántica y generación de código eficiente utilizando la infraestructura LLVM.

2 Descripción del Proyecto

2.1 URL del Repositorio

El proyecto se encuentra alojado en GitHub en la siguiente dirección: `https://github.com/AdrianSouto/hulk-compiler-cpp.git`

2.2 Características del Lenguaje HULK

HULK es un lenguaje que combina paradigmas funcional y orientado a objetos, incorporando las siguientes características:

- **Sistema de tipos estático** con inferencia automática
- **Funciones de primera clase** con soporte para closures
- **Programación orientada a objetos** con herencia simple
- **Expresiones condicionales** y estructuras de control
- **Operadores aritméticos, lógicos y de comparación**
- **Manejo de cadenas** con concatenación
- **Declaraciones let** para ámbitos locales
- **Verificación de tipos** en tiempo de compilación

3 Arquitectura del Compilador

3.1 Diseño General

Se implementó una arquitectura modular basada en el patrón Visitor, organizando el compilador en las siguientes fases:

1. **Análisis Léxico**: Implementado con un lexer nativo en C++
2. **Análisis Sintáctico**: Utilizando Bison para generar el parser
3. **Construcción del AST**: Árbol de sintaxis abstracta con jerarquía de nodos
4. **Análisis Semántico**: Verificación de tipos y contextos
5. **Generación de Código**: Emisión de LLVM IR

3.2 Estructura del Proyecto

La organización del código fuente se estructuró de la siguiente manera:

Listing 1.1. Estructura del proyecto

```
hulk-compiler-cpp/
|-- include/            # Archivos de cabecera
|   |-- AST/           # Nodos del arbol sintactico
|   |-- Expressions/   # Nodos de expresiones
|   |-- Statements/    # Nodos de declaraciones
|   |-- Visitors/      # Patron Visitor
|   |-- Context/       # Manejo de contextos
|   +-- Types/         # Sistema de tipos
|-- src/               # Implementaciones
|-- parser.y           # Gramatica Bison
|-- main.cpp           # Punto de entrada
+-- Makefile           # Sistema de construccion
```

4 Implementación de las Fases del Compilador

4.1 Análisis Léxico

Se desarrolló un analizador léxico nativo en C++ que reconoce todos los tokens del lenguaje HULK. La implementación se encuentra en `FlexCompatibleLexer.cpp` y maneja:

- Identificadores y palabras reservadas
- Literales numéricos (enteros y decimales)
- Cadenas de caracteres
- Operadores aritméticos, lógicos y de comparación
- Delimitadores y símbolos especiales

4.2 Análisis Sintáctico

Se utilizó Bison para generar el analizador sintáctico a partir de una gramática LALR(1). La gramática implementada en `parser.y` define:

- Precedencia y asociatividad de operadores
- Reglas de producción para todas las construcciones del lenguaje
- Acciones semánticas para construcción del AST
- Manejo de errores sintácticos

4.3 Árbol de Sintaxis Abstracta

Se diseñó una jerarquía de clases para representar el AST:

- `ASTNode`: Clase base abstracta
- `ExpressionNode`: Nodos de expresiones
- `StatementNode`: Nodos de declaraciones
- Clases específicas para cada construcción del lenguaje

4.4 Análisis Semántico

Se implementó un sistema de verificación semántica que incluye:

- **Verificación de tipos**: Compatibilidad en operaciones y asignaciones
- **Resolución de identificadores**: Verificación de declaración antes de uso
- **Verificación de contextos**: Ámbitos de variables y funciones
- **Validación de herencia**: Correcta implementación de jerarquías de tipos

4.5 Generación de Código LLVM

Se utilizó la infraestructura LLVM para generar código intermedio optimizable. La implementación del visitor `LLVMCodegenVisitor` maneja:

- Generación de instrucciones LLVM IR
- Manejo de tipos LLVM correspondientes
- Gestión de memoria y variables locales
- Generación de funciones y llamadas
- Optimizaciones básicas

5 Decisiones de Implementación

5.1 Patrón Visitor

Se eligió el patrón Visitor para la generación de código por las siguientes razones:

- **Separación de responsabilidades**: La lógica de generación de código se mantiene separada de la estructura del AST
- **Extensibilidad**: Facilita la adición de nuevos tipos de análisis sin modificar las clases del AST
- **Mantenibilidad**: Permite modificar la generación de código independientemente

5.2 Sistema de Tipos

Se implementó un sistema de tipos estático con las siguientes características:

- **Tipos primitivos**: Number, String, Boolean
- **Tipos definidos por el usuario**: Con soporte para herencia
- **Inferencia de tipos**: Reducción de anotaciones explícitas
- **Verificación en tiempo de compilación**: Detección temprana de errores

5.3 Manejo de Contextos

Se desarrolló un sistema de contextos anidados para manejar:

- Ámbitos de variables locales y globales
- Resolución de identificadores
- Verificación de tipos en diferentes contextos
- Manejo de herencia y polimorfismo

6 Justificación Teórica

6.1 Teoría de Compiladores

La implementación se basó en los principios fundamentales de la teoría de compiladores:

- **Gramáticas libres de contexto**: Para definir la sintaxis del lenguaje
- **Análisis LALR(1)**: Para el análisis sintáctico eficiente
- **Análisis semántico dirigido por sintaxis**: Integración de verificaciones durante el parsing
- **Representación intermedia**: Uso de LLVM IR como código intermedio

6.2 Sistemas de Tipos

Se implementó un sistema de tipos basado en:

- **Teoría de tipos de Hindley-Milner**: Para inferencia automática
- **Subtyping**: Para manejo de herencia
- **Verificación estática**: Para garantizar seguridad de tipos

7 Evaluación del Trabajo

7.1 Evaluación Cuantitativa

El compilador desarrollado presenta las siguientes métricas:

- **Líneas de código**: Aproximadamente 8,000 líneas de C++
- **Clases implementadas**: 45+ clases para nodos del AST
- **Casos de prueba**: 50+ archivos de prueba
- **Tiempo de compilación**: Menos de 1 segundo para programas típicos

7.2 Evaluación Cualitativa

Se evaluaron los siguientes aspectos cualitativos:

- **Correctitud**: El compilador genera código correcto para todos los casos de prueba
- **Robustez**: Manejo adecuado de errores sintácticos y semánticos
- **Mantenibilidad**: Código bien estructurado y documentado
- **Extensibilidad**: Arquitectura que facilita futuras mejoras

8 Logros y Limitaciones

8.1 Logros Principales

Se lograron los siguientes objetivos:

- **Compilador funcional completo**: Implementación de todas las fases
- **Sistema de tipos robusto**: Con verificación estática y inferencia
- **Generación de código eficiente**: Utilizando LLVM IR
- **Manejo de herencia**: Implementación correcta de POO
- **Detección de errores**: Mensajes informativos para el usuario

8.2 Limitaciones Identificadas

Se reconocen las siguientes limitaciones:

- **Optimizaciones limitadas**: Se depende principalmente de las optimizaciones de LLVM
- **Manejo de memoria**: No se implementó recolección de basura automática
- **Biblioteca estándar**: Conjunto limitado de funciones predefinidas
- **Depuración**: Falta de información de depuración en el código generado
- **Mensajes de error**: Algunos mensajes podrían ser más descriptivos

8.3 Propuestas de Mejora

Para futuras versiones se proponen las siguientes mejoras:

- **Optimizaciones adicionales**: Implementar pases de optimización específicos
- **Recolector de basura**: Integrar un sistema de gestión automática de memoria
- **Biblioteca estándar extendida**: Ampliar las funciones disponibles
- **Información de depuración**: Generar metadatos para depuradores
- **Mejores mensajes de error**: Implementar recuperación de errores y sugerencias
- **Soporte para módulos**: Permitir organización del código en módulos

9 Conclusiones

Se desarrolló exitosamente un compilador completo para el lenguaje HULK que implementa todas las fases tradicionales de compilación. La arquitectura modular basada en el patrón Visitor demostró ser efectiva para mantener el código organizado y extensible.

El sistema de tipos implementado proporciona verificación estática robusta mientras mantiene la flexibilidad necesaria para un lenguaje moderno. La integración con LLVM permite generar código eficiente y aprovechar las optimizaciones existentes.

Las limitaciones identificadas representan oportunidades de mejora que no afectan la funcionalidad core del compilador. El proyecto constituye una base sólida para futuras extensiones y mejoras.

La experiencia de desarrollo proporcionó conocimientos profundos sobre teoría de compiladores, sistemas de tipos, y técnicas de generación de código, cumpliendo con los objetivos educativos del proyecto.

10 Referencias

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. 2nd edn. Addison-Wesley, Boston (2006)
2. Appel, A.W.: Modern Compiler Implementation in C. Cambridge University Press, Cambridge (2002)
3. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization, pp. 75-86 (2004)
4. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
5. Donnelly, C., Stallman, R.: Bison: The Yacc-compatible Parser Generator. Free Software Foundation (2020)