Informe

Compilador HULK-In-RUST

Richard Alejandro Matos Arderí

Abraham Romero Imbert

Mauricio Sunde Jiménez

Grupo 311, Ciencia de la Computación

Facultad de Matemática y Computación

Universidad de La Habana

2025

Índice

1 Introduccion al Lenguaje Hulk

HULK (Lenguaje de la Universidad de La Habana para Compiladores) es un lenguaje de programación didáctico, con tipado seguro, orientado a objetos e incremental, diseñado para el curso de Introducción a los Compiladores en la carrera de Ciencias de la Computación en la Universidad de La Habana.

A grandes rasgos, HULK es un lenguaje de programación orientado a objetos, con herencia simple, polimorfismo y encapsulamiento a nivel de clase. Además, en HULK es posible definir funciones globales fuera del ámbito de cualquier clase. También es posible definir una única expresión global que constituye el punto de entrada del programa.

1.1 Características del lenguaje

La mayoría de las construcciones sintácticas en HULK son expresiones, incluyendo las instrucciones condicionales y los ciclos. HULK es un lenguaje de tipado estático con inferencia de tipos opcional, lo que significa que algunas (o todas) las partes de un programa pueden ser anotadas con tipos, y el compilador verificará la consistencia de todas las operaciones.

2 Ejecución del Proyecto

El proyecto HULK-Compiler-RS incluye un archivo `Makefile` que automatiza tareas comunes como la compilación, ejecución y limpieza del proyecto. A continuación se detallan los objetivos y su uso.

2.1 Objetivos del Makefile

- **compile**: Construye el proyecto usando `cargo` y mueve los archivos generados (`out.ll`, `ast.txt`) al directorio `hulk`.
- **execute**: Depende de `compile`. Mueve los binarios generados para distintas plataformas (`output_macos`, `output.exe`, `output_linux`) y los archivos generados al directorio `hulk`.
- **clean**: Elimina los artefactos de compilación generados por `cargo` y remueve el directorio `hulk`.

2.2 Variables del Makefile

- `TARGET_DIR`: Directorio donde se recopilan los archivos de salida (por defecto `hulk`).
- `CARGO_DIR`: Directorio que contiene el proyecto en Rust (por defecto `Compiler`).
- `OUT_LL`: Ruta del archivo de salida en formato LLVM IR.
- `OUTPUT_TXT`: Ruta del archivo de salida con el AST (Árbol de Sintaxis Abstracta).
- `TARGETMac`: Ruta de salida del binario para macOS.

- `TARGETWindows`: Ruta de salida del binario para Windows.
- `TARGETLinux`: Ruta de salida del binario para Linux.

2.3 Uso

Para ejecutar las tareas definidas en el `Makefile`, utiliza los siguientes comandos desde la terminal:

- `make compile`
  Compila el proyecto y recopila los archivos intermedios.
- `make execute`
  Compila y recopila los binarios y archivos de salida para todas las plataformas(Al ejecutar este comando también se generara un html con la documentación completa del código, en la terminal aparecera la url).
- `make clean`
  Elimina los artefactos de compilación y el directorio de salida.

3 Fase de Generación de Código

La fase de generación de código en el compilador HULK está implementada en el crate `codegen`. Esta etapa es responsable de transformar el Árbol de Sintaxis Abstracta (AST) en código intermedio LLVM IR, que puede ser posteriormente optimizado y traducido a código máquina para distintas plataformas.

3.1 Estructura General

Cada nodo del AST implementa el trait `Codegen`, el cual define el método `codegen` encargado de emitir el código LLVM correspondiente a ese nodo. Esto permite que la generación de código sea modular y extensible, ya que cada tipo de nodo (expresiones, declaraciones, tipos, funciones, etc.) conoce cómo traducirse a LLVM IR.

El crate `codegen` contiene:

- La estructura `CodegenContext`, que mantiene el estado global de la generación (código generado, tablas de símbolos, tipos, funciones, vtables, etc.).
- Utilidades para emitir instrucciones LLVM, gestionar variables temporales, ámbitos léxicos y nombres únicos.
- Métodos para registrar tipos, funciones, métodos y atributos, así como para manejar la herencia y el polimorfismo.

3.2 Funcionamiento del Trait Codegen

El trait `Codegen` define el método principal para la generación de código:

```
pub trait Codegen {
    fn codegen(&self, context: &mut CodegenContext) -> String;
}
```

Cada nodo del AST implementa este trait, de modo que al recorrer el árbol, se va generando el código LLVM de manera recursiva y estructurada. Por ejemplo:

- Un nodo de suma genera la instrucción LLVM para sumar dos valores.
- Un nodo de definición de función emite la cabecera, reserva espacio para los parámetros y genera el cuerpo de la función.
- Un nodo de acceso a miembro genera el código para acceder a un campo de una estructura.

3.3 Soporte para Orientación a Objetos

La generación de código soporta características orientadas a objetos como:

- Definición de tipos (clases) y sus atributos y métodos.
- Herencia simple y polimorfismo mediante vtables (tablas de métodos virtuales).
- Llamadas a métodos dinámicas usando el mecanismo de vtable y `getelementptr`.
- Inicialización de instancias y constructores personalizados.

3.4 Ventajas de la Arquitectura

- **Modularidad:** Cada nodo AST es responsable de su propia traducción a LLVM IR.
- **Extensibilidad:** Es sencillo agregar nuevos nodos o modificar la generación de código de los existentes.
- **Manejo de contexto:** El uso de `CodegenContext` permite gestionar correctamente ámbitos, tipos y símbolos durante la generación.
- **Compatibilidad:** El código generado es estándar LLVM IR, lo que permite aprovechar herramientas y optimizaciones existentes.

En resumen, la fase de generación de código del compilador HULK traduce el AST a LLVM IR de manera estructurada, modular y orientada a objetos, facilitando la portabilidad y optimización del código generado.