Hulkrc

Jossué Arteche
Darío Hernández
Francisco Préstamo
Facultad de Mátematica y Computación, Universidad de la Habana

18 de junio de 2025

Índice

Resumen

Este informe presenta el desarrollo de Hulkrc, un compilador implementado en Rust como proyecto final de la asignatura Compilación, correspondiente a la carrera de Ciencias de la Computación. Hulk es un lenguaje de programación orientado a objetos, con tipado estático e inferencia de tipos, diseñado con el objetivo de explorar de forma práctica los principales componentes de un compilador moderno.

El compilador realiza un análisis léxico y sintáctico eficiente mediante generadores propios de lexer y parser, seguido de un análisis semántico que incluye verificación de tipos, resolución de herencia y chequeo de accesos a miembros. Posteriormente, traduce los programas fuente a LLVM IR, permitiendo la generación de código máquina optimizado a través de la cadena de herramientas de LLVM.

El sistema es capaz de compilar programas generales que utilizan clases, funciones, herencia y estructuras de control comunes, sirviendo como base para experimentar con extensiones futuras y técnicas avanzadas de compilación.

1. El lenguaje Hulk

HULK (Havana University Language for Kompilers) es un lenguaje de programación diseñado con fines didácticos para la asignatura Compilación, impartida en la carrera de Ciencias de la Computación de la Universidad de La Habana. Se trata de un lenguaje orientado a objetos, con tipado estático seguro y soporte para inferencia de tipos, que permite explorar los conceptos fundamentales en el diseño e implementación de compiladores modernos.

Desde una perspectiva general, HULK combina características esenciales de los lenguajes orientados a objetos, como la herencia simple, el polimorfismo y la encapsulación a nivel de clases. Además, permite la definición de funciones globales fuera del ámbito de las clases, así como una única expresión global que actúa como punto de entrada del programa.

Una particularidad importante de HULK es que la mayoría de sus construcciones sintácticas son expresiones, incluyendo las instrucciones condicionales y los ciclos, lo cual otorga al lenguaje una gran flexibilidad expresiva. Gracias a su sistema de tipos estático con inferencia opcional, el lenguaje permite omitir anotaciones en partes del código sin perder seguridad, ya que el compilador se encarga de verificar la consistencia de los tipos en todas las operaciones.

2. Detalles de Implementación

La implementación del compilador de **HULK** se estructura en varias etapas clásicas del proceso de compilación, cada una con responsabilidades bien definidas y desarrolladas en el lenguaje de programación **Rust**. La elección de Rust no solo permite aprovechar sus garantías de seguridad y eficiencia en la gestión de memoria, sino que también facilita una arquitectura modular y robusta, a partir de crates, ideal para proyectos de mediana y gran escala como un compilador.

El proceso de compilación en HULK se divide en cuatro fases principales:

1. **Análisis léxico**: donde la secuencia de caracteres del programa fuente se transforma en una secuencia de *tokens* significativos para el lenguaje.

2. **Análisis sintáctico**: que toma los tokens y construye un árbol de sintaxis abstracta (*AST*), validando la estructura gramatical del programa.

3. **Análisis semántico**: encargado de verificar que el programa sea coherente a nivel de tipos, ámbitos, herencia y accesibilidad de miembros.

4. **Generación de código**: donde se traduce el AST validado a código intermedio *LLVM IR*, permitiendo posteriormente la compilación a código máquina optimizado.

Cada una de estas etapas fue implementada desde cero, buscando un equilibrio entre eficiencia, claridad y extensibilidad. A continuación, se describen en detalle los aspectos más relevantes del diseño y desarrollo de cada una de ellas.

2.1. Análisis Léxico

El análisis léxico del compilador de HULK fue implementado mediante un generador de analizadores léxicos desarrollado desde cero en *Rust*, inspirado en los conceptos presentados en el libro *Compilers: Principles, Techniques, and Tools* (también conocido como *el libro del dragón*).

Este generador permite construir analizadores léxicos a partir de expresiones regulares definidas para cada tipo de token. El motor de expresiones regulares implementado soporta operaciones fundamentales como concatenación, disyunción (`|`), repetición (`*`, `+`, `?`), rangos de caracteres (`[A-Z]`), y complementos (`[^A-Z]`), lo que permite definir de forma precisa patrones para palabras clave, identificadores, operadores, literales, entre otros.

Cada expresión regular es convertida inicialmente en un autómata finito no determinista (AFND). Luego, se construye un único autómata que combina todos los AFND correspondientes a los diferentes tokens. Para ello, se agrega un estado inicial común y se establece una prioridad entre los estados de aceptación, con el fin de resolver conflictos en casos de prefijos comunes entre diferentes tokens.

Posteriormente, este autómata combinado se transforma en un autómata finito determinista (AFD) mediante el algoritmo clásico de construcción por subconjuntos (*powerset construction*). El resultado es un autómata determinista optimizado y listo para la fase de tokenización.

Durante la ejecución, el analizador léxico simula el AFD sobre la entrada del programa, devolviendo siempre el token que corresponde al prefijo más largo reconocido. Esta estrategia, conocida como *maximal munch*, garantiza una desambiguación eficiente y precisa entre tokens que comparten prefijos.

Este enfoque permitió obtener un analizador léxico eficiente, extensible y alineado con las prácticas utilizadas en compiladores reales.

2.2. Análisis Sintáctico

Para el análisis sintáctico del compilador de HULK se desarrolló un generador de analizadores sintácticos **LALR** propio, optimizado y completamente escrito en *Rust*. Este generador fue diseñado para construir autómatas de forma eficiente, mediante la detección de *follows* espontáneos y su propagación a través de los *kernels* de los estados *LR(0)*, a partir de los cuales se calcula luego la clausura *LR(1)* final.

El generador soporta un sistema de *atributación* que permite asociar a cada producción gramatical una función de reducción. Estas funciones pueden devolver un valor de un tipo arbitrario parametrizable por el usuario (por ejemplo, `Option<i32>` en una calculadora), y reciben como argumentos los valores devueltos por las reducciones asociadas a los símbolos de la producción. Este enfoque está inspirado en herramientas como `yacc` y `lalrpop`, permitiendo que la definición de una gramática sea concisa y expresiva, sin necesidad de pasar por la creación de un árbol de derivación que luego tenga que ser convertido en un AST.

Con el objetivo de facilitar la definición y mantenimiento de las gramáticas, se implementó un **DSL** basado en macros declarativas. Este DSL permite expresar producciones gramaticales de manera sucinta y legible, mejorando significativamente la mantenibilidad del parser, la incorporación o eliminación de constructos gramaticales son extremadamente sencillos gracias a esto. Además, el sistema está diseñado para ser flexible mediante el uso de `traits`, permitiendo que el usuario defina su propio lexer e integrarlo con el generador de parser.

Gracias a este generador, se reimplementó el parser de HULK (originalmente construido con `lalrpop`) utilizando el nuevo DSL. La nueva versión logra los mismos resultados que la anterior, con niveles de mantenibilidad y legibilidad comparables con `lalrpop`.

2.3. Análisis Semántico

La etapa de análisis semántico en el compilador de HULK tiene como objetivo validar la corrección del programa a nivel de tipos, ámbitos, herencia y uso de miembros. Esta fase también prepara la información necesaria para la generación de código, incluyendo el sistema de tipos y la estructura jerárquica de las clases.

Se implementó un sistema de **tipado estático con inferencia local**, que permite al compilador deducir el tipo de expresiones complejas como estructuras condicionales (`if-else`) o listas literales, incluso cuando no se proporcionan anotaciones explícitas. Para determinar el tipo común de expresiones compuestas, se calcula el **ancestro común más cercano** (LCA, por sus siglas en inglés) dentro de la jerarquía de herencia, empleando un algoritmo eficiente basado en calcular el *minimun range query* en el *euler tour* del árbol de herencia usando *sparse tables*.

Durante esta fase también se realiza:

- La **verificación de definiciones** de variables y símbolos, garantizando que cada identificador utilizado haya sido previamente declarado en un ámbito válido.
- La **resolución de métodos y atributos**, comprobando su existencia y accesibilidad en la clase correspondiente o en alguna de sus superclases.
- El chequeo de **conformidad de tipos**, asegurando que se respete la compatibilidad en asignaciones, llamadas a funciones, argumentos y retornos.
- La validación de la **sobrescritura de métodos heredados**, verificando que las firmas coincidan de manera compatible y respeten las reglas de **varianza** (covarianza en el tipo de retorno y contravarianza en los parámetros).
- La inexistencia de ciclos en la herencia de los tipos. Para garantizar una jerarquía válida y robusta.

Para facilitar tanto esta etapa como la posterior generación de código, las definiciones de tipos se ordenan topológicamente en función de la relación de herencia. Esto permite que la información de clases padre esté disponible antes del análisis de sus clases derivadas y evita ciclos en la jerarquía.

Gracias a esta fase, se garantiza que los programas en HULK sean semánticamente válidos antes de ser traducidos a código intermedio, sentando las bases para una compilación correcta y segura.

2.4. Generación de Código

La etapa final del compilador de HULK consiste en la **traducción del árbol de sintaxis abstracta (AST)** validado a código intermedio en `LLVM IR`. Esta representación de bajo nivel es ampliamente utilizada en compiladores modernos por su portabilidad, flexibilidad y compatibilidad con optimizaciones avanzadas.

La estrategia de generación de código utilizada sigue un enfoque tradicional de **descomposición de expresiones** en operaciones más simples, mediante la creación explícita de variables temporales y el uso de instrucciones primitivas del lenguaje intermedio. Este proceso garantiza que cada subexpresión sea evaluada en el orden correcto, con resultados almacenados de forma intermedia y reutilizable.

A cada constructo del lenguaje fuente (como asignaciones, llamadas a métodos, estructuras condicionales, ciclos, etc.) se le asigna una secuencia correspondiente en `LLVM IR` que respeta tanto la semántica como los tipos inferidos previamente.

Además, se traduce la jerarquía de clases de HULK a estructuras anidadas de `LLVM`, y se manejan dinámicamente las tablas de métodos virtuales (v-tables) para soportar la invocación polimórfica y la herencia. Se asegura también que cada símbolo, ya sea variable local, global o atributo de clase, se mapee correctamente a su representación en memoria.

Este enfoque permite aprovechar el backend de LLVM para la optimización y generación de código máquina para múltiples arquitecturas, asegurando eficiencia sin sacrificar claridad en la implementación.

3. Manejo de Errores

Un componente esencial en la implementación del compilador de HULK es el **módulo de manejo de errores**, diseñado para proporcionar mensajes claros, precisos y contextualizados al usuario durante la compilación.

Este módulo comienza analizando el código fuente y registrando la posición de todos los *saltos de línea* en un arreglo. Esto permite realizar una **búsqueda binaria** eficiente sobre este arreglo para determinar, dada una posición absoluta en el texto, a qué línea pertenece un determinado carácter. Este enfoque garantiza tiempos de respuesta óptimos incluso en archivos fuente extensos.

Cada tipo de error semántico o sintáctico está representado mediante una estructura especializada que encapsula información relevante, incluyendo:

- La **posición** exacta del error (como el índice del carácter problemático).
- El **tipo de error** (por ejemplo, uso de variable no definida, error de tipo, invocación incorrecta, etc.).
- Un **mensaje explicativo** que describe el problema de forma amigable.

Una vez determinado el número de línea, se extrae el *slice* correspondiente al contenido de dicha línea para realizar un **formateo vistoso del mensaje**. Este formato incluye la línea de código original y un subrayado o puntero visual que señala con precisión el lugar del error, similar al estilo de mensajes que emiten compiladores modernos como `rustc` o `clang`.

Esta infraestructura permite que varios errores se reporten de forma clara y útil, ordenados segun su posicion en el código, facilitando la depuración por parte del programador y elevando la calidad general de la experiencia de desarrollo con HULK.

4. Bonus: Librería estándar

Como una característica extra del proyecto, se implementó una librería estándar que define funciones como `exp`, `sin`, etc. en esencia, todas las definidas en la definición del lenguaje, además se implementó la función floor para obtener la parte entera de un número, los cuales siempre son tratados como números de punto flotante. Estas funciones están en el scope de todos los programas de hulk compilados con hulkrc.

5. Conclusiones

El desarrollo de Hulkrc ha permitido poner en práctica los conceptos fundamentales de la teoría de compiladores, desde la construcción de analizadores léxicos y sintácticos hasta la implementación de un sistema de tipos robusto y la generación de código intermedio eficiente. La decisión de implementar generadores propios para el lexer y el parser, así como el diseño de un sistema de inferencia de tipos y manejo de errores avanzado, ha enriquecido significativamente la comprensión de los desafíos y soluciones en la construcción de compiladores modernos.

El uso de Rust como lenguaje de implementación ha aportado ventajas en términos de seguridad, modularidad y rendimiento, demostrando su idoneidad para proyectos de esta naturaleza. Además, la arquitectura modular adoptada facilita la extensión futura del compilador, permitiendo experimentar con nuevas características del lenguaje o técnicas de optimización.

Como resultado, Hulkrc no solo cumple con los objetivos académicos del proyecto, sino que también constituye una base sólida para el aprendizaje y la investigación en el área de compiladores.