Universidad de la República

Facultad de Ingeniería

Compilación de un lenguaje funcional simple a LLVM

Informe de Proyecto de Grado presentado por

Agustín Gazzano, Andrés Collares

como requisito de graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Alberto Pardo
Marcos Viera

Montevideo, 24 de septiembre de 2025

Agradecimientos

Queremos agradecer a todas las personas que nos ayudaron en el proceso de desarrollo de este proyecto.

Especialmente a nuestros tutores Alberto y Marcos por la disposición, compromiso y paciencia durante este largo proceso. Valoramos inmensamente el tiempo dedicado en llamadas, correos y constantes devoluciones y sugerencias para mejorar este trabajo.

También a nuestras familias y amigos por el apoyo y la motivación que nos ayudaron a seguir avanzando.

Resumen

Este proyecto de grado tiene como objetivo la implementación de un lenguaje de programación funcional utilizando Haskell como lenguaje de desarrollo y la infraestructura de compilación LLVM (*Low-Level Virtual Machine*). El trabajo se enfoca en la implementación de un compilador y un intérprete para un lenguaje funcional simple, los cuales generan código en el lenguaje intermedio de LLVM (LLVM IR). La optimización y generación de código máquina son delegadas a las herramientas provistas por LLVM.

El lenguaje funcional creado, denominado *LambdaKal*, basa su diseño en Kaleidoscope, que es el lenguaje de programación utilizado como ejemplo en la documentación oficial de LLVM. LambdaKal es un lenguaje tipado, conciso y expresivo, capaz de ejecutarse tanto a partir de archivos fuente como en un entorno interactivo *REPL (Read-Eval-Print Loop)* con un sistema de compilación en tiempo de ejecución o *JIT (Just-In-Time compilation)*.

LambdaKal cuenta con tres tipos de datos primitivos: enteros (`Int`), números en punto flotante (`Double`) y booleanos (`Bool`). Además, admite estructuras de datos como listas y tuplas, incluyendo tuplas anidadas. El lenguaje permite la definición de funciones, incluyendo funciones recursivas y de alto orden.

En cuanto al control de flujo, ofrece construcciones `if/else` y la definición de contextos locales mediante expresiones `let...in`. La biblioteca estándar proporciona un conjunto de funciones para la manipulación de listas y tuplas, así como operaciones sobre tipos escalares como seno, coseno, valor absoluto, etc. También se incluyen funciones típicas de los lenguajes funcionales modernos, tales como `map`, `filter`, y `foldr`.

Una característica destacada de *LambdaKal* es su capacidad para integrarse con funciones escritas en otros lenguajes como C, mediante el uso de bibliotecas compartidas dinámicas en formato `.so` (*Shared Object*). Esta funcionalidad permite extender el lenguaje y aprovechar funcionalidades de la biblioteca estándar de C como la función `printf`.

Gracias al uso de primitivas de bajo nivel provistas por LLVM, se logró una implementación eficiente, con un rendimiento que en ciertas pruebas iguala o incluso supera al de lenguajes funcionales establecidos como Haskell. Además, es posible interpretar y compilar para múltiples arquitecturas gracias al uso de *LLVM IR* como objetivo de compilación.

La arquitectura del compilador ha sido diseñada de manera modular para favorecer la extensibilidad y el mantenimiento del código. Asimismo, se desarrolló

una extensa suite de pruebas automatizadas de regresión que permite trabajar sobre el proyecto con mayor confianza y estabilidad.

Palabras clave: Haskell, LLVM, LambdaKal, Kaleidoscope, Programación funcional, Lenguaje de programación

Índice general

Capítulo 1

Introducción

A lo largo de las últimas décadas, los lenguajes de programación han experimentado una evolución significativa, tanto en su diseño como en las herramientas disponibles para su desarrollo. Han surgido lenguajes de alto nivel que facilitan la tarea de escribir programas, al tiempo que se han alcanzado avances en técnicas de optimización y generación eficiente de código ejecutable. En paralelo, se han desarrollado diversas herramientas que simplifican la construcción de nuevos lenguajes, permitiendo explorar otros enfoques en la definición de semánticas y sintaxis. Por ejemplo, en el ecosistema de Haskell (S.P. Jones, 2003), que es el lenguaje de programación funcional que es usado como lenguaje base en este proyecto, existen numerosas bibliotecas que permiten realizar análisis léxico y sintáctico. Ejemplo de esto es la biblioteca `Parsec` (Haskell, 2025c) que es usada extensamente.

En este proyecto se desarrolla el compilador de un lenguaje funcional simple llamado *LambdaKal*, que genera como salida código LLVM. LambdaKal basa parte de su diseño en el lenguaje Kaleidoscope (LLVM, 2025c). Parte de la documentación oficial de LLVM utiliza el lenguaje Kaleidoscope para explicar cómo implementar características que aparecen en lenguajes de programación en términos de LLVM. Kaleidoscope es un lenguaje imperativo sencillo, que se limita a números en punto flotante, definición de funciones, operadores, entre otros. LambdaKal es un lenguaje funcional con tipado simple que extiende las funcionalidades de Kaleidoscope, de las cuales las más notables son: soporte para múltiples tipos de datos, estructuras simples como tuplas o listas, funciones de alto orden y una biblioteca estándar.

LLVM se ha consolidado como una infraestructura de compilación poderosa. Es utilizado como *Backend* de compilación en múltiples lenguajes modernos como Haskell, Julia, Swift, Kotlin, Rust, Zig, entre otros. LLVM es una herramienta que resuelve un problema fundamental en la construcción de lenguajes: la estandarización y modularización de la generación de código de bajo nivel. LLVM proporciona un lenguaje intermedio (LLVM IR) junto con bibliotecas para generar, optimizar y emitir código máquina en múltiples arquitecturas.

El objetivo general del proyecto es construir un compilador para un lenguaje

puramente funcional, expresivo y eficiente, que combine las ventajas de Haskell como lenguaje de implementación con la capacidad de LLVM para generar código altamente optimizado y con soporte para múltiples arquitecturas. Para lograr esto, fue necesario realizar las siguientes tareas:

- Implementar un módulo que procesa la entrada de código fuente, la valide y genere un AST correspondiente.
- Generar el código de representación intermedia, LLVM IR. Esto se logra generando un AST propio de LLVM a partir del primer AST mencionado en el punto anterior.
- Agregar pasadas de optimización sobre el código intermedio, dependiendo del nivel de optimización elegido por el programador.
- Implementar un entorno interactivo (REPL) con soporte para compilación JIT. Este entorno debe soportar todas las funcionalidades del lenguaje.
- Permitir la interoperatividad con funciones escritas en otros lenguajes mediante bibliotecas compartidas.
- Desarrollar una biblioteca estándar que incluya operaciones sobre listas, tuplas y funciones de orden superior como `map`, `filter`, `foldr`, entre otras. Esta biblioteca será escrita en LambdaKal, para facilitar cambios o expansiones de la misma.
- Construir una suite de pruebas de regresión automatizadas, para garantizar el correcto funcionamiento del lenguaje al incorporar cambios.
- Evaluar el rendimiento de LambdaKal en comparación con otros lenguajes funcionales en modo REPL y compilado.

El resultado es un lenguaje modular y eficiente, capaz de ejecutarse tanto en modo REPL como en modo compilado. El uso directo de primitivas de LLVM permitió alcanzar un rendimiento competitivo. La arquitectura del compilador fue diseñada pensando en su mantenibilidad y extensibilidad, facilitando futuras mejoras o adaptaciones del lenguaje a dominios específicos.

En este documento se presentará el trabajo empezando por el marco teórico (Capítulo 2) que define las decisiones técnicas del proyecto, incluyendo una introducción a Haskell, LLVM y los conceptos fundamentales del diseño de compiladores. A continuación, en el capítulo 3, se describen las funcionalidades del lenguaje desde la perspectiva del usuario. El capítulo se divide en secciones según funcionalidades del lenguaje, partiendo desde los tipos de datos primitivos hacia estructuras y características más complejas. En el capítulo 4 se analiza la arquitectura interna del compilador, explicando las funciones que cumple cada parte del lenguaje y cómo se genera el código intermedio. En el capítulo 5 se abordan los aspectos técnicos complementarios como pruebas, rendimiento y comparación con otros lenguajes. Finalmente, se presentan las conclusiones generales del trabajo y posibles líneas de desarrollo futuro.

Capítulo 2

Revisión de antecedentes

2.1. Programación funcional

La programación funcional es uno de los grandes paradigmas de programación. A diferencia de otros paradigmas como la programación imperativa, donde las instrucciones alteran el estado de un programa, la programación funcional se basa en un modelo declarativo, donde el resultado de un programa se basa únicamente en las declaraciones de sus elementos como constantes, variables y funciones. En este sentido, las funciones son consideradas "ciudadanas de primera clase" ya que las mismas pueden ser usadas como parámetros o resultados de otras funciones, lo que se conoce como funciones de alto orden.

La programación funcional ofrece una visión de alto nivel de la programación, brindando a sus programadores una variedad de características que les ayudan a construir bibliotecas de funciones con sintaxis concisa pero poderosas y generales. La idea central de la programación funcional es el uso de funciones, particularmente funciones puras, que son aquellas cuyo resultado solo depende de los valores de sus entradas.

Las funciones puras garantizan que dado el mismo argumento, siempre producirán el mismo resultado, sin modificar variables externas ni interactuar con el estado del programa. Además, facilitan el razonamiento sobre el código, mejoran su reutilización y hacen que los programas sean más predecibles y fáciles de depurar. Permiten también optimizaciones como la memoización, donde los resultados de una función pueden ser almacenados y reutilizados sin necesidad de recalcularlos. (Simon Thompson, 2011).

Un ejemplo del poder expresivo y la versatilidad de la programación funcional es el uso de funciones de alto orden, como es el caso de `map`, que permiten aplicar transformaciones a los elementos de una colección de datos. Este tipo de funciones pueden utilizarse para modificar valores en una lista, procesar estructuras complejas o realizar operaciones sobre diferentes tipos de datos de manera eficiente y concisa.

La elegancia de la programación funcional es una consecuencia de la forma en

que se definen las funciones: en vez de definir una serie de pasos o instrucciones en el cuerpo de la función, se utiliza una ecuación para expresar el valor de una función dada una entrada arbitraria.

2.2. Haskell

En este proyecto se optó por usar Haskell (S.P. Jones, 2003) como lenguaje de implementación. Existen varias razones para elegir este lenguaje. En primer lugar, Haskell es un lenguaje funcional puro, lo que promueve un estilo de programación declarativo y modular. Esto permite describir la semántica del lenguaje de manera clara y concisa.

Además, Haskell cuenta con un sistema de tipos estático, lo que ayuda a detectar errores en tiempo de compilación; esto es valioso para la implementación de un lenguaje, ya que asegura una mayor robustez del sistema.

Otro punto a favor es la amplia disponibilidad de herramientas y bibliotecas especializadas en el desarrollo de lenguajes; en este proyecto se hace uso de bibliotecas de *parsing* como *Parsec* y *bindings* de LLVM como *llvm-hs* o *llvm-hs-pure* (Haskell, 2025a), (Haskell, 2025b).

Por último, Haskell es frecuentemente utilizado en entornos académicos y de investigación para prototipar lenguajes de programación, lo que refleja su idoneidad para este tipo de tareas y asegura una gran cantidad de recursos y ejemplos disponibles en la literatura.

2.3. Diseño de compiladores en tres fases

El diseño más popular para un compilador tradicional (como la mayoría de los compiladores de C) es el diseño de tres fases, cuyos componentes principales son el Frontend, el Optimizador y el Backend (ver Figura 2.1).

El Frontend analiza el código fuente, verificando errores y construyendo un Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés) específico del lenguaje para representar el código de entrada. Opcionalmente, el AST se convierte a una nueva representación intermedia para su optimización, y luego el Optimizador y el Backend procesan el código para generar el código final de bajo nivel (Amy Brown and Greg Wilson, 2011).

Figura 2.1: Compilador de tres fases

El Optimizador es responsable de realizar una amplia variedad de transformaciones para mejorar el tiempo de ejecución del código, como la eliminación

de cálculos redundantes, y suele ser más o menos independiente del lenguaje y del destino.

El Backend (también conocido como generador de código) luego asigna el código al conjunto de instrucciones del destino. Además de generar código correcto, es responsable de producir código eficiente que aproveche las características particulares de la arquitectura soportada. El resultado de este módulo es generar código en algún formato como ensamblador, código de máquina, *JVM bytecode*, etc.

Las partes comunes de un Backend de compilador incluyen la selección de instrucciones, la asignación de registros y la planificación de instrucciones.

La mayor ventaja de este diseño se presenta cuando un compilador decide admitir múltiples lenguajes fuente y/o múltiples arquitecturas como objetivo de compilación. Si el compilador utiliza una representación común del código en su Optimizador, entonces se puede escribir un Frontend para cualquier lenguaje que pueda compilar hacia esa representación, y un Backend para cualquier destino que pueda compilar desde ella. Ejemplo ilustrado en la Figura 2.2

Otra gran ventaja del diseño de tres fases es que los conocimientos y habilidades necesarios para implementar un Frontend son diferentes de los requeridos para el Optimizador y el Backend. Separar estos componentes facilita que una persona especializada en el Frontend pueda mejorar y mantener su parte del compilador de manera más eficiente.

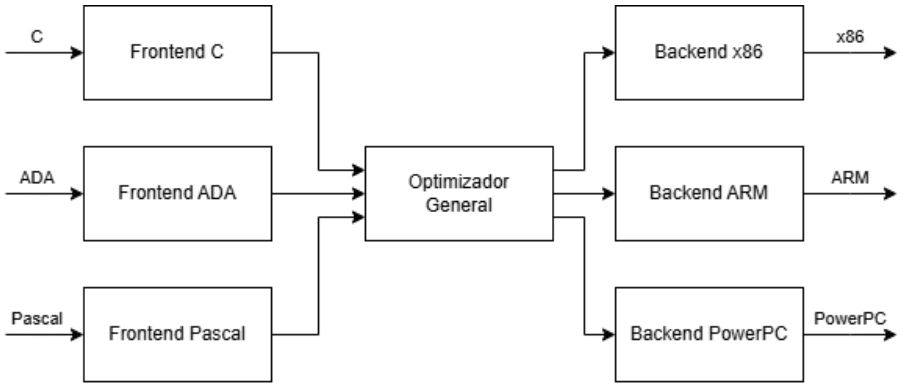

Figura 2.2: Compilador de tres fases con diferentes Frontends y Backends

2.4. LLVM

LLVM (Lattner y Adve, 2004) es una colección de tecnologías, herramientas y bibliotecas que facilitan la creación de compiladores en varias fases bien definidas, siguiendo el enfoque clásico de Frontend – Optimizador – Backend. Consiste en varios módulos entre los cuales los más importantes para este proyec-

to son la generación de código, optimización y compilación en tiempo real (JIT).

Para el diseño de compiladores de tres fases, LLVM provee herramientas para implementar el Optimizador y el Backend. Ofrece un conjunto robusto de optimizaciones predefinidas que mejoran el rendimiento del código generado sin necesidad de implementar técnicas de optimización manualmente. Del mismo modo, su backend soporta múltiples arquitecturas (x86, ARM, RISC-V, etc.), lo que permite compilar el lenguaje a diferentes plataformas con mínimos cambios.

Por estas razones, LLVM es una buena herramienta para implementar el Optimizador y el Backend en el diseño de tres fases. Entonces, deja al desarrollador enfocarse principalmente en la construcción del Frontend del compilador, es decir, el análisis léxico del lenguaje de entrada y la conversión a representación intermedia.

2.4.1. LLVM IR

El aspecto más importante del diseño de LLVM es la Representación Intermedia (*LLVM Intermediate Representation* o LLVM IR), que es una representación abstracta del código generado. LLVM IR está diseñado para realizar análisis y transformaciones de nivel intermedio, que se encuentran en la sección de optimización de un compilador. Fue creado con múltiples objetivos específicos en mente, incluyendo el soporte para optimizaciones ligeras en tiempo de ejecución, optimizaciones ínter-procedimientos o entre funciones, análisis de programa completo y transformaciones de reestructuración agresivas, entre otros.

Además, LLVM IR está definido como un lenguaje en sí mismo con una semántica bien establecida, formado por un conjunto de instrucciones virtual de bajo nivel, similar a un lenguaje ensamblador, con un mayor nivel de abstracción. LLVM IR admite secuencias lineales de instrucciones simples como *add, subtract, compare*. Estas instrucciones están en forma de *Código de tres direcciones*, lo que significa que toman un cierto número de entradas y producen un resultado en un registro diferente.

LLVM IR está estructurado según SSA (*Static single-assignment form*) (Appel, 1998), lo cual significa que cada variable es asignada únicamente una vez. El principal motivo para usar SSA es que permite simplificar y optimizar el código, especialmente cuando se tienen múltiples pasos de optimizaciones (Cytron, Ferrante, Rosen, Wegman, y Zadeck, 1991). Más adelante se verá cómo esto afecta a las operaciones de control de flujo que requieren que una variable tome valores diferentes según una condición.

LLVM es fuertemente tipado y utiliza un sistema de tipos simple (por ejemplo, i32 es un entero de 32 bits, i32** es un puntero a otro puntero a un entero de 32 bits). Además, algunos detalles específicos de la arquitectura están abstraídos. Por ejemplo, la convención de llamadas (Aaron Bloomfield, 2016) se maneja a través de las instrucciones call y ret con argumentos explícitos (LLVM, 2024).

Otra diferencia importante con el código de máquina es que LLVM IR no usa un conjunto fijo de registros con nombre, sino un conjunto no acotado de registros temporales, identificados con el prefijo `%` (Lattner y Adve, 2004).

En la Figura 2.3 se muestra un ejemplo de código LLVM IR, se definen 2 funciones: `half` y `main`, ambas retornan un valor de tipo `double`. La función `half` recibe un valor de tipo `double`, y lo divide entre el entero 2, antes de realizar esta división, transforma el entero 2 a `double` mediante `sitofp` y lo aloja en el registro `%1`. El resultado de la división se almacena en el registro `%2` y se devuelve como resultado de la función. Por otro lado, la función `main` asigna a su registro `%1` una suma de doubles (`fadd`) entre los números 1.0 y 2.0, para luego realizar una llamada a la función `half`, asignarla al registro `%2` y retornar ese valor.

```
define double @half(double %x_0) {
  %1 = sitofp i32 2 to double
  %2 = fdiv double %x_0, %1
  ret double %2
}

define double @main() {
  %1 = fadd double 1.000000e+01, 2.000000e+01
  %2 = call double @half(double %1)
  ret double %2
}
```

Figura 2.3: Ejemplo de código LLVM IR

2.4.2. Diferencias con otras herramientas de compilación

LLVM IR está bien especificado y es la única interfaz con el Optimizador. Esto significa que, para escribir un Frontend para LLVM, solo es necesario comprender qué es LLVM IR, cómo funciona y cuáles son sus invariantes.

Dado que LLVM IR tiene una representación textual, es posible construir un Frontend que genere una salida de LLVM IR en formato de texto y luego enviar esta salida al Optimizador y el generador de código de su elección. Como se verá en el capítulo 4, en este proyecto se completa el proceso de compilación mediante una llamada interna a un compilador de LLVM IR (*Clang*) cuando se quiere generar un archivo ejecutable como salida.

Esta es una característica bastante novedosa de LLVM y una de las principales razones de su éxito en una amplia variedad de aplicaciones. GCC, un compilador ampliamente utilizado y relativamente bien diseñado, no posee esta propiedad. Su representación intermedia, GIMPLE, no es completamente autónoma. Por ejemplo, cuando el generador de código de GCC emite información de depuración, debe retroceder y recorrer el AST del código fuente. GIMPLE usa una representación basada en tuplas para las operaciones, pero representa los operandos como referencias al AST del código fuente (Free

Software Foundation, 2025).

Las implicaciones de esto son que los desarrolladores de Frontends para GCC necesitan conocer y generar tanto las estructuras de datos en árbol de GCC como GIMPLE, y el Backend de GCC enfrenta problemas similares.

Finalmente, GCC no tiene una forma de generar una representación del código ni una manera de leer y escribir GIMPLE (y sus estructuras de datos relacionadas) en formato de texto. Como resultado, es relativamente difícil experimentar con GCC, lo que ha llevado a que tenga relativamente pocos Frontends.

Después del diseño de LLVM IR, el aspecto más importante de LLVM es que está construido como un conjunto de bibliotecas en lugar de un compilador monolítico como GCC o una máquina virtual cerrada como la *JVM* o *.NET*. LLVM es una infraestructura flexible que permite aplicar tecnología de compiladores a problemas específicos, como construir un compilador de C o un optimizador para fines especiales.

El Optimizador de LLVM funciona como una secuencia de pasadas independientes que transforman el código LLVM IR para mejorar su rendimiento. Dependiendo del nivel de optimización seleccionado, se ejecutan diferentes combinaciones de pasadas. Cada pasada está diseñada como una clase en C++, compilada en bibliotecas modulares, lo que permite seleccionar solo las optimizaciones necesarias para un proyecto específico.

Este diseño modular facilita la creación de herramientas personalizadas que aprovechan la tecnología de compiladores sin cargar funcionalidades innecesarias. Por ejemplo, un compilador JIT para procesamiento de imágenes puede elegir qué pasadas usar según las necesidades de su dominio y agregar optimizaciones específicas. Solo las pasadas utilizadas se vinculan a la aplicación final, reduciendo el tamaño y la complejidad del código.

Gracias a este enfoque basado en bibliotecas, LLVM es altamente adaptable y puede usarse en una amplia variedad de aplicaciones sin imponer restricciones innecesarias. Sin embargo, esta flexibilidad también lleva a que muchas personas malinterpreten su propósito, ya que LLVM en sí mismo no realiza ninguna tarea por defecto: depende del usuario decidir cómo aprovechar sus capacidades (Amy Brown and Greg Wilson, 2011).

2.4.3. Generación de código

Dentro de la arquitectura de tres fases descrita en la Sección 2.4, LLVM se ocupa principalmente del Optimizador y el Backend, dejando el desarrollo del Frontend a cargo de las diversas bibliotecas de terceros para varios lenguajes de programación.

En este proyecto, LLVM se utiliza como Backend y Optimizador del lenguaje. Todas las instrucciones ingresadas por el programador son traducidas al lenguaje de representación intermedia LLVM IR. Una vez logrado esto, las herramientas provistas por LLVM se encargan de optimizar la representación intermedia y compilar esta representación a un ejecutable o interpretarlo en tiempo real.

Entonces LLVM logra resolver los problemas de optimización automática del código y de ejecución en distintas arquitecturas, dejando como única tarea el desarrollo de un Frontend que pueda traducir instrucciones en un lenguaje arbitrario a la representación intermedia LLVM IR.

2.4.4. Optimización

Uno de los aspectos más interesantes de LLVM son las funcionalidades para optimizar el código IR. LLVM aplica transformaciones sobre el código IR para mejorar el rendimiento del programa, reducir su tamaño, o ambas cosas. Las optimizaciones abarcan aspectos de alto nivel como eliminación de código muerto, propagación de constantes, inlining de funciones, optimización de bucles hasta transformaciones más complejas como utilizar instrucciones específicas del hardware, vectorización u optimización de funciones recursivas mediante *tail call elimination*. Esto último resulta de mucha utilidad para implementar lenguajes funcionales, donde se hace mucho uso de la recursión (Wegman y Zadeck, 1991) (Muchnick, 1998).

2.4.5. Compilación en tiempo real (JIT)

La compilación JIT (*Just-In-Time*) permite ejecutar programas directamente desde el IR sin necesidad de generar un archivo binario precompilado. Esta funcionalidad de LLVM es conveniente para implementar una funcionalidad de REPL (*read–eval–print loop*).

2.5. Kaleidoscope

Kaleidoscope es un lenguaje de programación imperativa sencillo utilizado como ejemplo en la documentación oficial de LLVM (LLVM Tutorials, 2024). Se trata de un lenguaje imperativo que cuenta con un único tipo de datos: números en punto flotante de 64 bits.

Entre sus características fundamentales se encuentran:

- Definición de funciones y llamadas a funciones externas.
- Soporte para operadores sobre números en punto flotante con distintos niveles de precedencia.
- Estructura de control de flujo (`if`).
- Definición de variables locales mediante la construcción `let`.

En la documentación oficial de LLVM se detalla cómo construir un Frontend para Kaleidoscope utilizando el lenguaje C.

2.5.1. Implementación de Kaleidoscope en Haskell

Este proyecto está basado en el Tutorial/blog de Kaleidoscope en Haskell de Stephen Diehl (Stephen Diehl, 2024), llamado *Implementing a JIT Compiled Language with Haskell and LLVM*. En dicho trabajo se implementa el lenguaje imperativo Kaleidoscope utilizando Haskell y LLVM.

LambdaKal se basa en el prototipo de Kaleidoscope creado en este trabajo, con la adición de nuevos tipos de datos primitivos como enteros y booleanos, y la inclusión de estructuras de datos de tipos lista y tupla, y funcionalidades pertinentes a éstas.

A los efectos de que el lenguaje utilice un paradigma puramente funcional, no se implementaron funcionalidades presentes en Kaleidoscope como ciclos `for` o `while` y mutación de variables.

También se actualizó toda la infraestructura subyacente a nuevas versiones de LLVM y bibliotecas relacionadas de Haskell.

2.6. Uso de LLVM en otros lenguajes

LLVM es una infraestructura de compilación que ha tenido un impacto significativo en el desarrollo de lenguajes de programación modernos. Fue diseñado originalmente como un conjunto de herramientas para la optimización y compilación de los lenguajes C y C++. En la actualidad, numerosos lenguajes de programación compilan a LLVM IR directamente o lo utilizan como backend para generar código máquina. Algunos ejemplos destacados incluyen Haskell, Julia, Swift, Kotlin, Rust y Zig, entre otros.

LLVM se encuentra en constante evolución, la comunidad es activa y dinámica, realizando actualizaciones frecuentes, incorporación de nuevas funcionalidades y mejoras en el rendimiento. El ecosistema de LLVM también incluye herramientas complementarias como Clang 2.6.1 y herramientas de depuración como LLDB.

2.6.1. Clang

Clang es un Frontend de compilación de la familia de lenguajes de C (C, C++, Objective C/C++, OpenCL, y CUDA), que forma parte del proyecto LLVM. Está pensado como una alternativa a GCC completamente compatible. Además de ser un Frontend de compilación, también actúa como "compiler driver", con lo cual puede realizar el proceso completo de compilación llamando a las bibliotecas del Backend de LLVM (Anastasia Stulova and Sven van Haastregt, 2019). En el contexto de este proyecto, Clang es utilizado para generar los ejecutables una vez generado el código LLVM IR.

2.7. LLVM y Haskell

Existen bibliotecas que permiten trabajar con LLVM directamente desde Haskell, particularmente en este proyecto se usan *llvm-hs* y *llvm-hs-pure*. Estas bibliotecas aportan una interfaz (*bindings*) para crear instrucciones en LLVM IR y manejar el estado del árbol de sintaxis.

La biblioteca *llvm-hs* provee *bindings* a un nivel de abstracción similar a la biblioteca oficial de C de LLVM, y también funcionalidades extras, por ejemplo, interfaces monádicas para simplificar la generación de código LLVM IR a partir de un árbol de sintaxis. Estas bibliotecas son utilizadas por lenguajes como *Dex* (Adam Paszke, 2021).

Capítulo 3

LambdaKal

3.1. Descripción

LambdaKal es un lenguaje basado en la sintaxis de Kaleidoscope, con la particularidad de ser un lenguaje funcional.

El tipado es estático, siempre que se define una función debe especificarse cuales son los tipos de los parámetros y cual es el tipo de retorno. Por otro lado, admite sobrecarga de operadores, y el chequeo de tipos es estático, lo que significa que se realiza en tiempo de compilación.

LambdaKal es interpretado, el código fuente puede ser ejecutado en forma directa. Para ejecutar programas del lenguaje, se puede proveer un archivo de código fuente para interpretar, o correr una sesión interactiva o REPL en la cual se pueden ingresar comandos de a uno a la vez y se obtienen resultados en tiempo real. Adicionalmente, es posible compilar programas a un archivo ejecutable.

En cuanto a la sintaxis, LambdaKal no requiere indentación para definir bloques (*off-side rule*), aunque se pueden usar por motivos de estilo. Se usa el carácter ";" para delimitar expresiones.

Soporta tres tipos de datos básicos:

- Números enteros (`Integer`).
- Números en punto flotante (`Float`).
- true o false (`Boolean`).

Admite dos estructuras de datos que son las tuplas (pares ordenados) y las listas (cadena de valores del mismo tipo).

LambdaKal también permite definir funciones recursivas y funciones de alto orden.

3.2. Características

A continuación se explican en mayor detalle las principales características del lenguaje y su sintaxis; en el capítulo 4 se profundizará en la generación de código. Para todos los ejemplos se muestra una sesión en el REPL, las entradas del usuario están precedidas por `ready>`, mientras que las líneas sin ningún prefijo son resultados.

3.2.1. Integer, Float y Boolean

Los literales de tipo Integer, Float se definen utilizando los estándares de Haskell, y los booleanos mediante las palabras reservadas `true` y `false`.

Integer Los números enteros se pueden expresar en decimal, octal o hexadecimal. Sin embargo, el resultado de una expresión entera siempre se muestra en decimal. Son enteros con signo: se admiten representaciones negativas.

```
ready> 35;
35
ready> -0xFF;
-255
ready> 0o77;
63
```

Float Los números en punto flotante pueden ser expresados con separador decimal o en notación exponencial *(E notation)*.

```
ready> 1.50;
1.500000
ready> 7.51E-3;
0.007510
```

Boolean Los booleanos se expresan mediante los literales `true` y `false`, siempre en minúscula. Para los booleanos existen los operadores típicos como `&&` (AND), `||` (OR), `!` (NOT) y `^^` (XOR).

```
ready> true && false;
false
```

3.2.2. Operadores

Existen operadores unarios (prefijos) y binarios (infijos) predefinidos en el lenguaje. Los operadores descritos en esta sección funcionan sobre los tres tipos de datos básicos del lenguaje. En otra sección se profundiza sobre operadores primitivos definidos para los tipos estructurados (tuplas y listas).

```
ready> 2 + 3;
5
ready> !true;
false
ready> true || false;
true;
```

El lenguaje realiza sobrecarga de operadores, es decir que ciertos operadores están definidos para varios tipos de datos y su implementación interna depende de los tipos de datos utilizados, particularmente los operadores aritméticos que se pueden usar con `Integer` o `Float` indistintamente:

```
ready> 2 + 3.0;
5.000000
ready> 10.0 / 3;
3.333333
ready> 10 / 3;
3
ready> 10 % 3.0;
1.000000
```

Siempre que se utilice al menos un número en punto flotante en una operación aritmética, LambdaKal realiza coerción automática de Integer a Float en caso de ser necesario, y el resultado es un número en punto flotante.

Una característica presente en el lenguaje original Kaleidoscope, era la capacidad de definir operadores. Si bien esto se pudo implementar en LambdaKal, en la presente versión esto se eliminó en favor de reducir la complejidad del código, sobre todo en la resolución de los nombres de las funciones, ya que los operadores definidos por el usuario estaban definidos mediante funciones.

3.2.3. If/else

If es una expresión condicional que provee LambdaKal. Se trata de un operador ternario que devuelve un valor, en lugar de ser únicamente una instrucción de control de flujo.

```
ready> if 9 > 8 then 1 else 2;
1
ready> if false then 1 else 2;
```

```
2
-- someCondition(y) == false
ready> let x = (if someCondition(y) then 5 else 12) in x + 1;
13
```

`if` debe siempre tener una cláusula `else` dado que siempre debe poder evaluarse a un valor.

3.2.4. Funciones

Las funciones se definen mediante un nombre, una lista de parámetros tipados y un tipo de retorno. El retorno de las funciones es implícito, es decir, no existe la palabra `return` en el lenguaje, y el cuerpo de la función debe ser una expresión.

Luego de definir la función, puede ser invocada con sus parámetros entre paréntesis, no es necesario indicar el tipo de los parámetros en la invocación, como sí lo es en la definición.

```
ready> def add(int a, int b) -> int: a + b;
ready> def sub(int a, int b) -> int: a - b;
ready> add(5, sub(6, 2));
9
```

Las funciones pueden ser redefinidas:

```
ready> def f(int a) -> double: a + 1.0;
ready> def f(int a) -> double: a + 1000.0;
ready> f(3);
1003.000000
```

Un aspecto a tener en cuenta es que el lenguaje maneja un binding dinámico de funciones (también conocido como late binding). Esto significa que cuando se invoca a una función, siempre se va a usar su última definición, o sea que en caso de tener implementaciones anteriores, las mismas quedan obsoletas. Esto implica que al redefinir una función, también se puede alterar el comportamiento de otras funciones que la invoquen. En el siguiente ejemplo se definen dos funciones sencillas: `f1` que devuelve 1 y `f2` que devuelve `2 * f1`. En primera instancia `f2` devuelve el valor 2, pero si cambiamos la implementación de `f1` a que devuelva 10, entonces `f2` devolverá 20.

```
ready> def f1() -> int: 1;
ready> def f2() -> int: 2 * f1();
ready> f2();
2
-- redefinicion de f1
ready> def f1() -> int: 10;
ready> f2();
20
```

Es posible ejecutar funciones de forma recursiva. En el siguiente ejemplo se muestra una función que calcula la suma de todos los números entre 0 y el número provisto.

```
ready> def sum(int a) -> int:
    if (a == 0) then 0 else a + sum(a - 1);
ready> sum(10);
55
```

Otro ejemplo de recursión es el cálculo de la serie Fibonacci mediante una recursión de segundo orden, en la Figura 3.1.

```
def fib(int x) -> int:
   if x < 3 then
      1
   else
      fib(x-1) + fib(x-2);

fib(6);
-- resultado 8
```

Figura 3.1: Código de la función `fib` implementada con recursión de segundo orden.

También es posible implementar funciones con recursión mutua, esto es, un par de funciones que están definidas una en términos de la otra. Para esto también se hace uso de la capacidad de LambdaKal de redefinir funciones, para dar una definición no recursiva a una de las funciones temporalmente. Esto es necesario dado que de lo contrario se produce un error al no poder identificar esa función en el contexto del programa.

```
ready> def isOdd(int n) -> bool: false; -- definicion temporal
ready> def isEven(int n) -> bool: if n == 0 then true else
↪  isOdd(n - 1);
ready> def isOdd(int n) -> bool: if n == 0 then false else
↪  isEven(n - 1);
55
ready> isOdd(9);
true
```

Al ser un lenguaje funcional, la recursión es la principal forma de ejecutar secuencias de código o iterar sobre estructuras de datos. No se cuenta con estructuras **for** o **while** como en otros lenguajes.

3.2.5. Listas

Las listas permiten crear secuencias de tipo homogéneo de cualquiera de los tres tipos primitivos del lenguaje (Integer, Float y Boolean); la lista está delimitada por paréntesis rectos y sus elementos separados por comas.

A diferencia de las tuplas, todos los elementos dentro de la lista deben ser del mismo tipo.

```
ready> const a = 99;
ready> [1, 2, a];
[1, 2, 99]
ready> [2.0, 4.7, 11.1];
[2.000000, 4.700000, 11.100000]
ready> [true, true && false, true];
[true, false, true]
ready> def foo(int x) -> int: x * 10;
ready> [foo(2), 1 + foo(1), 44, foo(foo(1))];
[20, 11, 44, 100]
```

El lenguaje provee funcionalidades básicas para manipular listas, por ejemplo `head` para obtener el primer elemento de la lista y `tail` para obtener la cola de la lista.

```
ready> head [5, 10, 7];
5
ready> tail [5, 10, 7];
[10, 7]
```

El operador para insertar elementos al principio de la lista es ":", es un operador de tipo binario cuyo primer argumento es un escalar y el segundo una lista.

```
ready> 6:[7, 8];
[6, 7, 8]
ready> 6:(1:[7, 8]);
[6, 1, 7, 8]
```

Para realizar transformaciones sobre listas se puede hacer uso de las funciones de la biblioteca estándar, por ejemplo las funciones `map`, `filter` o `reverse`.

`map` recibe una lista como primer parámetro y el nombre de una función booleana como segundo parámetro. Genera una nueva lista aplicando la función sobre cada elemento.

```
ready> def triple(int x) -> int: x*3;
ready> map_int([5, 10, 7], triple);
[15, 30, 21]
```

`filter` recibe una lista como primer parámetro y el nombre de una función como segundo parámetro. Genera una nueva lista que solamente incluye los elementos que al aplicarles la función devuelven `true`.

```
ready> def even(int x) -> bool: x % 2 == 0;
ready> filter_int([2,3,10,8,9], even);
[2, 10, 8]
```

`reverse` genera una nueva lista con los elementos en el orden inverso a la original.

```
ready> reverse_int([3,4,5,6]);
[6, 5, 4, 3]
```

Estos son algunas de las funciones de la biblioteca, más adelante en la sección 3.3 se lista el detalle de las funciones existentes. LambdaKal también da la posibilidad de definir funciones que reciban listas como parámetro o que devuelvan listas.

3.2.6. Tuplas

La estructura de datos más básica en LambdaKal son las tuplas. Son pares ordenados de expresiones de cualquier tipo. Para definir una tupla se utilizan paréntesis curvos con sus dos elementos separados por comas. Las tuplas no son necesariamente homogéneas, por ejemplo, se puede definir una tupla en la cual el primer elemento sea un entero y el segundo punto flotante. Las tuplas también admiten tuplas como componente, con lo cual se puede formar una especie de anidación de elementos de distintos tipos; en el caso de que todos los elementos tengan el mismo tipo, es preferible utilizar listas, como se muestra en 3.2.5.

En este ejemplo, se muestran ejemplos de tuplas con tipos de datos mixtos (integer y double).

```
ready> (9, 1.0);
(9, 1.0)
ready> (9, (4.5, 10));
(9, (4.5, 10))
```

Las tuplas también permiten tener listas como elemento.

```
ready> (45.0, [1,2,3]);
(45.000000, [1, 2, 3])
ready> ([1,2,3], [4.5, 6.7]);
([1, 2, 3], [4.500000, 6.700000])
```

Los operadores `fst` y `snd` proveen una manera de acceder al primer y segundo componente de la tupla, respectivamente.

```
ready> fst (9, 10);
9
ready> snd (9, 10);
10
ready> snd (9, (4.5, 10));
(4.5, 10)
```

Para mantener la sintaxis sencilla y reducir la complejidad del lenguaje, se optó por no implementar *pattern matching*, por lo cual esta es la única forma de acceder a los elementos de las tuplas.

3.2.7. Let

Let permite definir un contexto con variables locales, la sintaxis permite definir varios valores a la vez separándolos con comas. A diferencia de las constantes, estas variables no son accesibles fuera del contexto en el cual son definidas. *Let* es una construcción sintáctica del lenguaje que permite construir una expresión con un contexto local. En el último ejemplo se ve cómo puede ser usado como componente de otra expresión.

```
ready> let pi = 3.14 in pi*4;
12.560000
ready> let a = 1, b = 2, c = 3 in
    a + b + c;
6
ready> 10 + let x = 40 in x + 15;
65
```

Asimismo, se puede escribir `let` de forma anidada.

```
ready> let a = 1 in
    let b = 2.0 in
        let c = 3 in
            a + b + c;
6.000000
```

Los valores definidos en `let` no necesariamente deben ser primitivos, pueden ser también estructuras o llamadas a funciones.

```
ready> let list = [2,3,4] in 1:list;
[1, 2, 3, 4]
ready> let log10 = log(10.0) in log10 + 90;
92.302585
```

3.2.8. Funciones de alto orden

Se puede definir funciones de alto orden que reciben una o más funciones como parámetros. Al momento de hacer la llamada a una función de alto orden, todas las funciones involucradas deben ser definidas de antemano, ya que no existen funciones anónimas como en Haskell u otros lenguajes (Simon Thompson, 2011). El tipo de una función pasada como parámetro tiene tipo `fun (<params>) -><returnType><name>`. Para poder invocar la función se usa el nombre (`name`) dentro de la función de alto orden.

En este ejemplo, se define una función sencilla que permite aplicar una función de **double** a **double** sobre un número entero **d**. Para esto, suma 0.0 a **d**, aprovechando que esta operación transforma el resultado a double; luego aplica la función **g** que recibe como parámetro.

```
ready>
def floater(
    fun (double) -> double g,
    int d
) -> double:
    g(d + 0.0);
ready> def half(double x) -> double: x / 2;
ready> floater(half, 3);
1.500000
```

Otro ejemplo son las funciones de manipulación de listas de la biblioteca estándar, por ejemplo `filter`, en la Figura 3.2 se muestra la definición y el uso de esta función.

```
ready>
def filter_int([int] l, fun (int) -> bool f) -> [int]:
  if l == [] then
    []
  else
    (if f(head l) then
      head l : filter_int(tail l, f)
    else
      filter_int(tail l, f));

ready> def isEven(int n) -> bool: n % 2 == 0;
filter_int([1,2,3,4,5], isEven);
```

Figura 3.2: función `filter_int` de la biblioteca estándar

3.2.9. Funciones externas

LambdaKal provee una interfaz para invocar funciones que estén enlazadas mediante bibliotecas compartidas del ejecutable. Las funciones externas son raramente usadas por el programador, pero internamente son usadas para imprimir resultados o generar estructuras de datos; esto se explica en la sección de biblioteca estándar (3.3).

La utilidad de esta directiva es que el programa pueda interactuar con objetos definidos en otros lenguajes, por ejemplo, C.

En el siguiente ejemplo, se declara la función `putchari`, que recibe un entero e imprime el carácter correspondiente en la tabla ASCII del mismo. El cuerpo de esta función está definido en un archivo C, que es compilado a un archivo .so, que luego es enlazado al programa previo a su ejecución.

```
ready> extern putchari(int i) -> int;
ready> putchari(71);
GO
-- G es el char 71 en ASCII, el valor de retorno del programa es
↪  0.
```

3.3. Biblioteca estándar

El lenguaje posee una biblioteca estándar con funciones de uso general. Estas funciones están accesibles al programador en todo momento, tanto al compilar como en el REPL, sin necesidad de incluirlas en el programa con una directiva como en otros lenguajes como C.

La biblioteca estándar tiene tres partes principales:

Por un lado, existen módulos escritos en C que permiten al lenguaje realizar ciertas funciones básicas, como imprimir en pantalla o alojar espacio en memoria para listas. Estas funciones están disponibles al momento de ejecución mediante archivos **so** "Shared Object". Estos archivos .so deben ser incluidos en el directorio `/usr/lib/liblambdakal.so` para que el ejecutable pueda enlazarse dinámicamente en tiempo de ejecución.

En segundo lugar, existen funciones auxiliares definidas en LLVM IR de manera directa, usando los *bindings* de Haskell para LLVM. Este es el caso de las funciones de conversión de tipos, por ejemplo `int_to_double` o `double_to_int`.

Por último, la mayoría de las funciones de manipulación de listas están definidas en el mismo lenguaje en una colección de archivos con código fuente de LambdaKal. Estos archivos son leídos al momento de ejecutar el intérprete y todas las definiciones presentes en ellos son agregadas al contexto de ejecución.

3.3.1. Interfaz con funciones de C

Como se mencionó anteriormente, LambdaKal admite la utilización de funciones externas definidas en C o en cualquier lenguaje capaz de compilar a archivo ELF (Executable and Linkable Format), en formato .so (shared object).

Estas funciones proveen las funcionalidades de más bajo nivel al lenguaje, algunas de ellas son:

- Impresión en pantalla de todos los tipos del lenguaje (`bool, int, float, tuple, list`). Se hace uso de la función `printf` de C en todos los casos.
- Generación de nodos de listas: funciones como *_alloc_int_list_node*

En el módulo `io`, algunas de las funciones más importantes son:

- `printil`, `printdl`, `printbl` - Reciben un parámetro de tipo lista de `int`, `double` o `bool` respectivamente, e imprime todos los elementos en pantalla entre paréntesis rectos y separados por comas.

En el módulo `list`, se definen las funciones `_alloc_int_list_node`, `_alloc_double_list_node` y `_alloc_bool_list_node`. Estas funciones son usadas internamente por el lenguaje al momento de instanciar nodos de listas.

3.3.2. Funciones y tipos auxiliares

Las funciones auxiliares son en su mayor parte definiciones base para poder hacer interfaz con las funciones de C. A estos efectos, para cada función definida en C existe una declaración o *stub* de dicha función para que el intérprete de LLVM IR pueda referenciarlas por nombre.

También define funciones que no forman parte de los módulos en C de LambdaKal, pero que existen en el contexto del proceso del intérprete JIT de LLVM,

este es el caso de las funciones `sin` y `cos`. Cuando en el IR se invoca a una función que no tiene un cuerpo definido, invoca a la función mediante `dlsym` (LLVM Tutorials, 2024). `dlsym` se usa en sistemas basados en Unix para obtener la dirección de una función o variable dentro de una biblioteca cargada dinámicamente (Linux man pages, 2024).

Dentro del mismo contexto existen declaraciones para las estructuras de datos que utiliza LambdaKal y que son referenciadas internamente por el intérprete de LLVM IR. Este es el caso para las definiciones de los tipos `IntList`, `FloatList` y `BoolList`, que son construidas en este módulo.

Finalmente, en este módulo también se definen ciertas funciones que hacen uso directo de las instrucciones de LLVM, como es el caso de `int_to_double` o `double_to_int`. `int_to_double` hace uso de la instrucción `sitofp` (*Signed integer to Floating Point*) para transformar un entero a punto flotante. Para reducir la complejidad de la sintaxis, `sitofp` no tiene operando equivalente en la sintaxis de LambdaKal, sino que es expuesto exclusivamente a través de esta función.

A continuación, una lista no exhaustiva de las funciones provistas por esta parte de la biblioteca.

- Funciones matemáticas
  - `sin`
  - `cos`
  - `log`
  - `fabs`
  - `rand`
- Funciones de conversión de tipos
  - `int_to_double`
  - `double_to_int`

3.3.3. Funciones escritas en código fuente

La mayor parte de la biblioteca de LambdaKal está implementada con código fuente de LambdaKal, con lo cual resulta muy fácil extenderla. La biblioteca estándar es cargada automáticamente como preámbulo tanto en el intérprete como en el REPL. No es necesario usar una directiva para incluir funciones de la biblioteca.

La biblioteca estándar provee principalmente funciones para trabajar con listas. Las funciones descritas a continuación son algunas de las disponibles en la biblioteca estándar. Cada una tiene dos versiones, una con sufijo `_int` y otra con sufijo `_double`. Se explicará su funcionalidad de forma genérica para ambos tipos.

- `length` - Cantidad de elementos en la lista.

- `reverse` - Devuelve una copia de la lista revertida.
- `any` / `all` - Recibe una lista de un tipo T y una función de tipo `(T) ->bool`, devuelve si la condición se cumple para alguno o todos los elementos respectivamente.
- `map` - Recibe una lista y una función, devuelve una copia de la lista en la cual sus elementos son el resultado de la aplicación de la función sobre cada elemento de la lista.
- `filter` - Recibe una lista de tipo [T] y una función de tipo `(T) ->bool`, devuelve una copia de la lista en la cual solo están presentes los elementos que evalúan a `true` en la función.
- `nth` - devuelve el elemento en la posición n (empezando por 0). Asume que la lista tiene largo menor a n.
- `foldl` - Recibe una lista, un valor inicial y una función de dos parámetros, devuelve un valor singular. Aplica la función sobre el valor inicial y el primer elemento de la lista, luego aplica la función entre el resultado y el segundo elemento de la lista, y repite este proceso hasta llegar al final de la lista.
- `foldr` - Recibe una lista, un valor inicial y una función de dos parámetros, devuelve un valor singular. Aplica la función sobre el valor inicial y el último elemento de la lista, luego aplica la función entre el resultado y el penúltimo elemento de la lista, y repite este proceso hasta llegar al principio de la lista.

3.4. Archivos de código fuente

Aparte del intérprete en modo REPL, se puede interpretar el código desde un archivo, para esto se utiliza la bandera `-f` seguida del nombre del archivo.

La Figura 3.3 muestra un ejemplo de un programa que puede ser utilizado como entrada desde un archivo.

Para ejecutar este programa, es posible hacerlo mediante el siguiente comando en la consola.

```
cabal run lambdakal -- -f ./test/programs/primes.k -q

[47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2]
```

3.5. Compilación a un ejecutable

Una necesidad fundamental del lenguaje es poder compilar el código a un archivo ejecutable, en código de máquina, que pueda correr desde la línea de comandos sin invocar al intérprete del lenguaje.

```
def divisibleAny(int n, int i) -> bool:
    if i < 2 then
        false
    else
        if n % i == 0 then
            true
        else
            divisibleAny(n, i - 1);

def isPrime(int n) -> bool: !divisibleAny(n, n - 1);

def primesUntil(int n) -> [int]:
    if n < 2 then
        []
    else
        if isPrime(n) then
            n:primesUntil(n - 1)
        else
            primesUntil(n - 1);

primesUntil(50);
```

Figura 3.3: Programa que devuelve una lista de números primos menores a 50

En este caso se provee un archivo fuente, con el comando `-f` seguido del nombre del archivo y la opción `-c` o `compile` para generar el ejecutable a partir de dicho archivo.

```
cabal run lambdakal -- -f ./test/programs/recursive_fib.k -c
```

Se generarán 2 archivos dentro del directorio `/test/programs`:

- recursive_fib.ll - Contiene el código LLVM IR generado a partir del código fuente.
- recursive_fib - Es el archivo ejecutable.

Una vez generado este último archivo, se puede ejecutar el programa mediante el comando `./recursive_fib`. El resultado de la última instrucción de la función main se imprime en pantalla.

```
./recursive_fib.o
8
```

3.6. Intérprete (REPL)

Uno de los objetivos del proyecto es lograr que el lenguaje tenga un intérprete, es decir, dar la posibilidad al usuario de iniciar un entorno o sesión donde se pueda escribir las instrucciones de a una a la vez y obtener resultados en tiempo real. El intérprete genera código LLVM IR, al igual que el compilador regular. Para esto, el lenguaje se apoya sobre la compilación JIT que provee la biblioteca de LLVM.

El lenguaje está implementado de tal forma que el código puede ser generado de a bloques, para ir generando un código intermedio a medida que el programador ingrese definiciones. Estas definiciones son agregadas a un contexto que persiste a lo largo de la sesión y son accesibles al momento de ingresar cualquier expresión.

Entonces es posible definir funciones o constantes, ver en tiempo real que la sintaxis sea correcta, para luego evaluar una expresión y obtener un resultado en la consola.

Cuando se ingresa una expresión o definición, el contexto vuelve a generarse desde cero, esto es necesario porque se debe tener en cuenta todas las definiciones anteriores al momento de evaluar otra definición o expresión.

Cada vez que se ingresa una expresión, se genera nuevamente la función **main**, que es la que el intérprete de LLVM invoca cada vez que se quiere correr el programa.

El intérprete extiende la sintaxis del lenguaje original para agregar ciertas funcionalidades para el programador. Por ejemplo, la directiva `:l <filename>` permite cargar un archivo al REPL, con lo cual se hacen accesibles todas sus definiciones.

```
ready> :l test/programs/primes.k;
ready> isPrime(7);
true
```

3.7. Opciones de línea de comandos

El ejecutable de LambdaKal acepta parámetros por línea de comandos para modificar su funcionamiento; estos parámetros son:

`optimizationLevel` Define el nivel de optimización de LLVM, en el rango de 0 a 3.

`inputFile` Define el archivo que contiene el código fuente a ser procesado. Si no se especifica esta opción, se ejecuta el intérprete en modo REPL *(Read-Eval-Print-Loop)*

`emitLLVM` Indica si se imprime en pantalla la representación intermedia del código LLVM producido por el intérprete.

`emitDefs` Indica si se imprime en pantalla el AST de LLVM IR de la última definición generada en el programa. Esto es útil para diagnosticar errores

producidos en tiempo de ejecución en LLVM IR.

`emitAST` Indica si se imprime en pantalla el AST de LambdaKal correspondiente al programa.

`failOnErrors` Indica si el programa debería detener su ejecución cuando se encuentra un error. Esta opción es `false` cuando se ejecuta en modo REPL.

`compile` Se usa en conjunto con `inputFile` para generar un archivo ejecutable a partir del código fuente.

```
$ cabal run lambdakal -- -help
LambdaKal compiler

Usage: lambdakal [-o|--opt-level OPT] [-f|--file FILE]
↪  [-l|--llvm] [-a|--ast]
                [-d|--defs] [-e|--fail-on-errors]
                ↪  [-c|--compile]

  LambdaKal compiler

Available options:
  -o,--opt-level OPT       Optimization level 0-3
  -f,--file FILE           File to read from
  -l,--llvm                Show LLVM IR output
  -a,--ast                 Show AST representation
  -d,--defs                Show LLVM Module definitions
  -e,--fail-on-errors      Fail on errors
  -c,--compile             Compile to native code
  -h,--help                Show this help text
```

Por ejemplo, para correr el intérprete en modo REPL con máximo nivel de optimización y sin imprimir la representación intermedia LLVM, se invoca de la forma:

```
cabal run lambdakal -- -o 3 -q
```

Capítulo 4

Proceso de compilación

4.1. Arquitectura

En esta sección se describe la arquitectura de compilación/interpretación del lenguaje, la cual se divide en módulos. Cada módulo se encarga de un paso en el proceso de interpretación del programa fuente, generando un resultado para ser utilizado por el siguiente módulo. En el caso de ejecutar en modo REPL, este proceso se repite hasta que se finalice la sesión del REPL. En la Figura 4.1 se ve un diagrama de la arquitectura que describe el proceso de compilación / interpretación a partir de una entrada de código fuente. En las siguientes secciones se describirá cada uno de los componentes de la arquitectura que se ven en la figura.

4.1.1. Parser/Lexer

En esta parte se toma la entrada del usuario, ya sea desde la consola o desde un archivo y se realiza la tokenización y *parsing*. Esto está implementado utilizando la biblioteca Parsec de Haskell (Haskell, 2025c). En la implementación desarrollada se realiza un chequeo de sintaxis, pero no de tipos; esta funcionalidad podría ser implementada, pero se optó por enfocar el desarrollo en la compilación a LLVM y las partes funcionales del lenguaje.

Sintaxis

La gramática del lenguaje está conformada por 2 grandes categorías sintácticas: Expresiones y Declaraciones. Una expresión puede ser evaluada a un valor concreto, mientras que las declaraciones son definiciones globales del entorno de ejecución del programa.

Como resultado del *parsing* de expresiones se generan arboles de sintaxis que luego se pueden interpretar.

En términos formales, la sintaxis viene dada por la reglas de producción mostradas en el Cuadro 4.1.

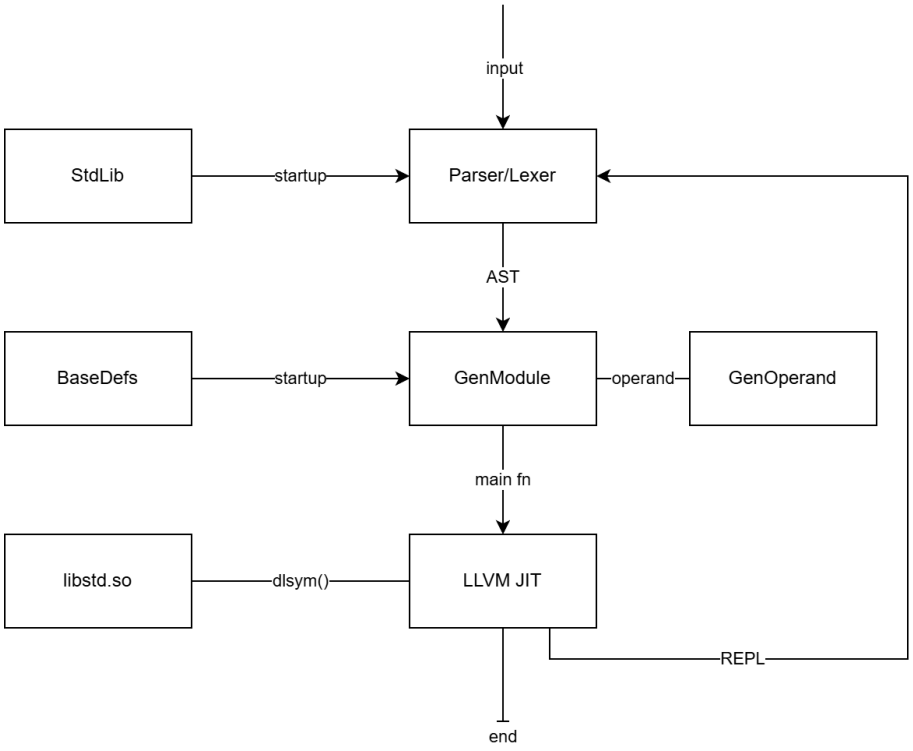

Figura 4.1: Arquitectura de LambdaKal

S	TopLevel
TopLevel	Expr | Declaration
Type	double | int | bool | tuple(Type,Type) | [Type] | fun (CommaSepType) ->Type
Expr	i | f | b | Tuple | List | Let | Var | Call | If | UnaryOp | BinaryOp
Tuple	(Expr , Expr)
List	[CommaSepExpr]
Let	let Name = Expr in Expr
Var	Name
Call	Name (CommaSepExpr)
If	if Expr then Expr else Expr
UnaryOp	Unary Expr
BinaryOp	Expr Binary Expr
Unary	! | - | fst | snd | head | tail
Binary	* | / | + | - | < | > | <= | >= | == | != | ^^ | && | ||
Declaration	Function | Extern | Constant | TypeDef
Constant	const Name Int | const Name Float | const Name Bool
Function	def Name (CommaSepName) ->Type : Expr
CommaSepType	Type | Type , CommaSepType
CommaSepExpr	Expr | Expr , CommaSepExpr
CommaSepName	Name | Name , CommaSepName

Cuadro 4.1: Reglas de generación de la gramática del lenguaje.
$i \in \mathbb{Z}, f \in \mathbb{R}, b \in \{true, false\}$. Name es cualquier cadena alfanumérica que no comienza en número.

El resultado de este módulo es un AST que representa el programa en su código fuente original. En la Figura 4.2 se muestran los Data Types correspondientes al AST de LambdaKal.

La estructura de este AST puede tener como nodo raíz una expresión o una declaración. Las declaraciones quedan guardadas en el contexto para referenciar posteriormente, mientras que las expresiones son evaluadas inmediatamente. En todo caso, el AST es enviado al siguiente módulo para generar el código intermedio correspondiente.

Cuando se ejecuta en modo REPL, se pueden hacer múltiples invocaciones a este módulo, conservando el contexto. Si se ejecuta LambdaKal con un archivo fuente como entrada, el módulo procesa toda la entrada en un solo paso.

A modo de ejemplo, suponiendo que el programador ingresa `x * 2`, el módulo Parser genera un AST en el cual `2` es un `Int`, `x` es un `Var` y `*` es un `BinOp`. Esto forma una expresión simple que según el Data Type se representa como `(BinOp (Name "*") (Var (Name "x")) (Int 2))`.

Con el ejemplo anterior, se define una expresión que corresponde a multipli-

```
data TopLevel
  = Expr Expr
  | Declaration Declaration
  deriving stock (Eq, Ord, Show)

data Expr
  = Int Integer
  | Float Double
  | Bool Bool
  | TupleI Expr Expr
  | List [Expr]
  | Let Name Expr Expr
  | Var Name
  | Call Name [Expr]
  | If Expr Expr Expr
  | UnaryOp Name Expr
  | BinOp Name Expr Expr
  deriving stock (Eq, Ord, Show)

data Declaration
  = Function Name [(Type, ParameterName)] Type Expr
  | Extern Name [(Type, ParameterName)] Type
  deriving stock (Eq, Ord, Show)

data Type
  = Double
  | Integer
  | Boolean
  | Tuple Type Type
  | ListType Type
  | FunType [Type] Type
  deriving stock (Eq, Ord, Show)
```

Figura 4.2: Data Types utilizados para representar los AST de LambdaKal

car una variable de nombre “x” por el número 2. Si uno intenta ingresar esto sin contexto previo en el REPL, se produce un error, dado que se espera que la variable x esté definida. Se puede proveer el contexto necesario a esta expresión de varias formas, por ejemplo utilizando `Let` o definiendo `x` como parámetro de una función que englobe la expresión.

Esta última opción se puede ver en la Figura 4.3, donde primero se muestra el AST generado a partir de una definición de una función y luego la utilización de esta función en una expresión. Se puede observar cómo una definición de función genera un constructor del tipo `Declaration`, mientras que una expresión genera un constructor `Expr`. El cuerpo de la función consiste de la expresión utilizada de ejemplo anteriormente: `(BinOp (Name "*") (Var (Name "x")) (Int 2))`

```
def foo(int x) -> int: x * 2;
Declaration (Function (Name "foo") [(Integer,ParameterName "x")]
↪  Integer (BinOp (Name "*") (Var (Name "x")) (Int 2)))]

foo(9) + 2;
Expr (BinOp (Name "+") (Call (Name "foo") [Int 9]) (Int 2))
```

Figura 4.3: AST de LambdaKal correspondiente a la definición de una función y una expresión. La representación impresa del AST del programa se puede obtener pasando el parámetro -a de LambdaKal.

4.1.2. GenOperand

GenOperand genera operandos para ser utilizados en el AST correspondiente a LLVM IR. Específicamente, se generan las instrucciones individuales que luego se usan para construir una definición completa en GenModule. Entonces este módulo recibe como entrada un AST de LambdaKal, restringido a Expresiones únicamente (nunca Definitions), y devuelve como resultado un AST de LLVM para una expresión.

Continuando con el ejemplo de `x * 2`, el fragmento del AST generado se puede ver en la Figura 4.4. Observar que se le asigna un número a esta expresión (UnName 1), esto es lo que luego se muestra como el identificador `%1` en la representación impresa de LLVM IR. Tener en cuenta que la expresión `x * 2` no es válida si `x` no existe en el contexto, en este caso supondremos que `x` fue definido anteriormente en algún nodo superior del AST.

4.1.3. GenModule

Dada una representación del AST generada por el Parser, el módulo GenModule se encarga de generar un módulo LLVM ya utilizando el AST correspondiente a LLVM IR. Los módulos de LLVM son el contenedor de nivel superior de

```
UnName 1 := Mul {operand0 = LocalReference (
    IntegerType {typeBits = 32}) (Name "x_0"),
    operand1 = ConstantOperand (
        Int {integerBits = 32, integerValue = 2}
)}
```

Figura 4.4: Ejemplo simple de un fragmento del AST de LLVM IR para la entrada `x * 2`

todos los demás objetos de LLVM IR. Cada módulo contiene: variables globales, funciones, bibliotecas y diversos datos sobre las características del objetivo de compilación.

En este proceso se debe construir desde cero un árbol de sintaxis en LLVM IR traduciendo las definiciones del Parser a sus equivalentes y agregando definiciones donde sea necesario.

Aparte de las definiciones provenientes del parser, GenModule recibe las definiciones iniciales de StdLib y de BaseDefs, las cuales agrega al contexto del módulo.

En este módulo se generan únicamente definiciones globales de funciones o constantes, apoyándose en el módulo GenOperand para generar las expresiones que pertenezcan al cuerpo de una función.

Otro objetivo importante de este módulo es generar la función de nombre `main`, que es la que el JIT de LLVM utiliza como punto de partida para la ejecución. Esta función es redefinida y ejecutada cada vez que el programador ingresa una expresión. Como último paso de la función `main`, `GenModule` agrega la llamada a la función `printf` en la IR para poder visualizar el resultado, esto se puede ver en la Figura 4.6.

4.1.4. StdLib

Este módulo es una colección de archivos fuente de LambdaKal. Estos archivos contienen definiciones de funciones que forman parte de la biblioteca estándar, estas definiciones son procesadas por el Parser e inyectadas en el contexto antes de interpretar la entrada del programa, de esta forma las funciones están disponibles al programador en todo momento, ya sea en modo REPL o Interpretando desde un archivo. Un ejemplo de este tipo de funciones es map, cuya definición se ve en la Figura 4.7.

```
GlobalDefinition (
    Function {returnType = IntegerType {typeBits = 32},
        name = Name "foo",
        parameters = ([
            Parameter (IntegerType {typeBits = 32}) (Name "x_0")
        ], False),
    basicBlocks = [
        BasicBlock (UnName 0) [
        UnName 1 := Mul {operand0 = LocalReference (
            IntegerType {typeBits = 32}) (Name "x_0"),
            operand1 = ConstantOperand (
                Int {integerBits = 32, integerValue = 2}
        )}
    ] (
        Do ( Ret {returnOperand = Just (
                LocalReference (IntegerType {typeBits = 32})
                ↪  (UnName 1)
            ), metadata' = []})
    )]})
```

Figura 4.5: AST (simplificado) del módulo LLVM IR correspondiente a la función `foo` de la Figura 4.3

```
def map_int([int] l, fun (int) -> int f) -> [int]:
  if l == [] then
    []
  else
    f(head l) : map_int(tail l, f);
```

Figura 4.7: Código fuente LamdaKal de la función map_int

4.1.5. BaseDefs

BaseDefs es un conjunto de definiciones de funciones y también de tipos que forman parte del contexto de ejecución. En este caso, no son procesadas por el Parser sino que están definidas directamente en el AST de LLVM IR utilizando los *bindings* para Haskell de la biblioteca llvm-hs. La razón por la cual existe este módulo es porque hay tipos, operandos y funciones que forman parte de

```
-- 1 + foo(9);
define i32 @main() {
  %1 = call i32 @foo.11(i32 9)
  %2 = add i32 1, %1
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
  ↪  ([4 x i8], [4 x i8]* @fmtStr, i32 0, i32 0), i32 %2)
  ret i32 0
}
```

Figura 4.6: LLVM IR de una función `main`

la biblioteca estándar pero que no pueden ser definidas utilizando la sintaxis de LambdaKal como en `StdLib`. Algunos ejemplos de esto son definiciones de funciones matemáticas básicas como `sin` o `cos`, o funciones de entrada/salida como `printf` para imprimir valores de tipo `int, float` o `boolean`.

LLVM IR no incorpora implementaciones intrínsecas de funciones de la biblioteca estándar del lenguaje C, como `printf` o funciones matemáticas, pero sí permite declaraciones de funciones externas como se vió anteriormente.

Estas funciones son resueltas en tiempo de enlace o de ejecución por el enlazador del sistema, el cual vincula el programa generado con la biblioteca estándar de C disponible en el entorno de destino (`libc` en Unix).

Por otro lado, en este módulo también se declaran las funciones de conversión de tipos como `int_to_double`. En este caso, no es una declaración externa, sino que es una función con implementación definida dentro del módulo del programa. Esta función utiliza operandos de LLVM IR que no tienen equivalente en la sintaxis de LambdaKal, por lo cual se hace uso los *bindings* de `llvm-hs` directamente (Figura 4.8).

4.1.6. libstd.so

Este es el archivo compilado a un objeto dinámico a partir de los archivos de código fuente en C de la biblioteca de LambdaKal, es la parte de la biblioteca estándar que provee interfaz con funciones de C, como se describió en capítulos anteriores. Este archivo existe en la ubicación por defecto `/usr/lib` y es necesario que exista para correr LambdaKal.

Si bien las funciones definidas en estos módulos están disponibles al entorno de ejecución del programa, no están pensadas para ser utilizadas directamente por el programador, sino que son utilizadas por el lenguaje internamente.

`libstd.so` es un binario en formato ELF (Executable and Linkable Format), este tipo de archivos contienen código máquina, tablas de símbolos y secciones de relocación. Este archivo puede ser enlazado en tiempo de ejecución para exponer sus funciones al programa LambdaKal.

```
GlobalDefinition
  functionDefaults
    { name = Name (fromString "int_to_double"),
      parameters = ([Parameter (IntegerType {typeBits = 32})
      ↪  (Name "x_0") []], False),
      returnType = FloatingPointType DoubleFP,
      basicBlocks =
        [ BasicBlock
            (UnName 0)
            [ UnName 1
                := I.SIToFP
                  { operand0 = LocalReference (IntegerType
                  ↪  {typeBits = 32}) (Name "x_0"),
                    I.type' = FloatingPointType
                    ↪  {floatingPointType = DoubleFP},
                    I.metadata = []
                  }
            ]
            (Do (Ret {returnOperand = Just (LocalReference
            ↪  (FloatingPointType {floatingPointType =
            ↪  DoubleFP}) (UnName 1)), I.metadata' = []}))
        ]
    },
```

Figura 4.8: Función int_to_double definida mediante los *bindings* de `llvm-hs`. `SIToFP` es la instrucción de LLVM utilizada para convertir de int a punto flotante.

4.1.7. LLVM JIT

Una vez creado el AST de LLVM IR, el módulo JIT se encarga de correr las transformaciones necesarias para optimizar el código y finalmente ejecutarlo con MCJIT de LLVM (LLVM, 2025b).

4.2. Generación de código intermedio

Como se vió anteriormente, el lenguaje se divide en Definiciones y Expresiones. Las definiciones se traducen a declaraciones en LLVM, que se guardan en un contexto al cual otras definiciones o expresiones pueden acceder. Por otro lado, cuando se encuentra una expresión, se genera una función `main` que es interpretada por el JIT de LLVM. Si la función `main` ya existe (esto puede suceder en el REPL) entonces es redefinida y solo se ejecuta la última versión del `main`.

Para cada tipo de expresión reconocida por LambdaKal, se define una función que genera el LLVM IR correspondiente.

A continuación, se explica cómo se genera el código intermedio para cada una de las funcionalidades y características del lenguaje.

4.2.1. Tipos estáticos

LambdaKal es un lenguaje de tipado estático. Al momento de la traducción a LLVM IR se asume que el AST representa una expresión o declaración bien tipada.

LLVM IR utiliza también un tipado estático (LLVM, 2024), entonces solo hizo falta implementar un mapeo de los tipos de LambdaKal a tipos de LLVM IR al momento de generar instrucciones.

En la siguiente tabla se representa el mapeo de los tipos en el AST de LambdaKal contra la representación en LLVM IR.

LambdaKal	LLVM IR	Concepto
double	double	Punto Flotante de 64 bits
int	i32	Entero con signo de 32 bits
bool	i1	Entero de 1 bit

4.2.2. Operadores

LLVM provee instrucciones para realizar operaciones aritméticas y lógicas básicas, distinguiendo entre operaciones entre enteros y entre punto flotante. Por ejemplo, para sumar dos números enteros existe la instrucción `add`, mientras que para sumar dos números en punto flotante existe la instrucción `fadd`. Al momento de generar el código LLVM IR, el módulo *GenOperand* debe decidir cual de las dos instrucciones debe llamar para la suma, dado que LambdaKal implementa sobrecarga de operadores para utilizar el mismo operando para sumas entre enteros que para sumas entre números en punto flotante.

```
; ready> 10 / 3;
%1 = udiv i32 10, 3
; %1 = 3

-------

; ready> 10.0 / 3.0;
%1 = fdiv double 1.000000e+01, 3.000000e+00
; %1 = 3.3333333333333335

-------

; ready> 10.0 / 3;
%1 = sitofp i32 3 to double
%2 = fdiv double 1.000000e+01, %1
; %2 == 3.3333333333333335
```

Figura 4.9: Operador división (`/`) aplicado a distintos tipos

Para permitir utilizar tipos mixtos en operandos, por ejemplo la suma entre un *Integer* y un *Float*, LambdaKal introduce una instrucción de conversión de tipo como paso intermedio en el código LLVM IR.

Pese a que el tipado es estático, los tipos de los operandos pueden influir en la traducción del AST a instrucciones en LLVM IR. En la Figura 4.9, se ven tres ejemplos de esto, en primer lugar la división entre `int`, que devuelve un `int`; luego la división entre `double`, que devuelve un `double`; y finalmente la división de un `double` y un `int`, que también devuelve `double`.

Por otro lado, también existen operadores de LambdaKal que no tienen correspondencia directa con operadores de LLVM, como por ejemplo el operador de concatenación de un elemento al principio de la lista (`:`). En este caso, el operador se traduce a una serie de instrucciones que agregan un nuevo nodo a la lista. La función del modulo GenOperand que logra esto se puede ver en la Figura 4.10

4.2.3. Funciones

Las funciones en LLVM IR consisten de un nombre, tipo de retorno, parámetros con tipo y cuerpo de la función. Dado que las funciones en LambdaKal deben tener un tipado explícito en su valor de retorno y en sus parámetros, la conversión de funciones a LLVM IR es bastante simple. GenModule genera la firma de la función, mientras que la expresión del cuerpo es generada por el módulo GenOperand.

```
genOperand (S.BinOp ":" a b) localVars = do
  opA <- genOperand a localVars
  opB <- genOperand b localVars
  firstElem <- createListNode opA
  prependNode firstElem opB
```

Figura 4.10: Función que genera las instrucciones en LLVM IR para el operador ":".

```
1 declare %IntList* @map_int(%IntList*, i32 (i32)*)
2
3 define %IntList* @map_int.4(%IntList* %l_0, i32 (i32)* %f_0) {
4   ; cuerpo de la función
5 }
```

Figura 4.11: LLVM IR Generado para la función `map_int` de la biblioteca estándar de LambdaKal. Se han omitido los detalles de implementación del cuerpo de la función para este ejemplo.

En la Figura 4.11 se observa el código LLVM IR generado por LambdaKal para la función de la biblioteca estándar `map_int`. La primera línea es la firma de la función, en esta firma se puede observar que:

- internamente el nombre de la función es `@map_int`
- su tipo de retorno es `%IntList*`
- su primer argumento es una lista (`%IntList*`)
- su segundo argumento es un puntero a una función con tipo de retorno `i32` y un argumento de tipo `i32` (expresado como `i32 (i32)*`)

Luego de la firma de la función, viene el bloque del cuerpo. La firma inicial se usa internamente para determinar el tipo de la función cuando ocurre una llamada recursiva en su cuerpo.

4.2.4. Tuplas

Para generar una tupla se hace uso del tipo `struct` de LLVM IR. Este `struct` contiene un valor por cada componente de la tupla, que puede ser de tipo `i1`, `i32`, `double`, puntero a lista o puntero a tupla.

Los `struct` en LLVM permiten formar estructuras de cualquier tipo de dato, incluyendo punteros. Por esto resulta sencillo iterar sobre el AST de LambdaKal e ir generando las instrucciones necesarias para popular el `struct`.

Partiendo de un caso simple, tomaremos de ejemplo la tupla `(9, 1.0)`. En este ejemplo, en el registro `%1` se asigna espacio para un struct de tipo `{ i32, double }`, en el registro `%2` se obtiene el puntero al lugar izquierdo de la tupla y en la siguiente instrucción se guarda el número 9 en la dirección indicada por dicho puntero. Las siguientes dos instrucciones hacen lo mismo para el lado derecho de la tupla, con el valor `double 1.0`.

```
define i32 @main() {
  %1 = alloca { i32, double }, align 8
  %2 = getelementptr { i32, double }, { i32, double }* %1, i32
↪  0, i32 0
  store volatile i32 9, i32* %2, align 4
  %3 = getelementptr { i32, double }, { i32, double }* %1, i32
↪  0, i32 1
  store volatile double 1.000000e+00, double* %3, align 8
  ret i32 0
}
```

Supongamos ahora que se tiene una tupla dentro de otra: `(1, (2.0, true))`. En este caso la lógica es similar, si se ve la estructura de la tupla como un árbol, entonces la construcción de la tupla se realiza desde los nodos inferiores hacia los nodos superiores. Por lo cual el primer elemento generado es la tupla inferior `(2.0, true)`, se obtiene un puntero a la misma que se guarda en el lugar derecho de la tupla superior. De las líneas `2` a `6` se construye la tupla inferior, y en la línea `11` se asigna al espacio derecho de la tupla superior. Es fácil ver como esta lógica se extiende a cualquier combinación de estructuras de tuplas.

```
define i32 @main() {
  %1 = alloca { double, i1 }, align 8
  %2 = getelementptr { double, i1 }, { double, i1 }* %1, i32 0,
↪  i32 0
  store volatile double 2.000000e+00, double* %2, align 8
  %3 = getelementptr { double, i1 }, { double, i1 }* %1, i32 0,
↪  i32 1
  store volatile i1 true, i1* %3, align 1
  %4 = alloca { i32, { double, i1 }* }, align 8
  %5 = getelementptr { i32, { double, i1 }* }, { i32, { double,
↪  i1 }* }* %4, i32 0, i32 0
  store volatile i32 1, i32* %5, align 4
  %6 = getelementptr { i32, { double, i1 }* }, { i32, { double,
↪  i1 }* }* %4, i32 0, i32 1
  store volatile { double, i1 }* %1, { double, i1 }** %6, align
↪  8
  ret i32 0
}
```

La inclusión de una lista como elemento de una tupla se logra de forma muy similar. En lugar de tener un puntero a otro `struct`, se usa el puntero pre definido `%IntList*` (en caso de lista de enteros). El siguiente fragmento de código se generó a partir de la entrada `ready>(1, [1]);`

```
define i32 @main() {
  %1 = call %IntList* @_alloc_int_list_node()
  %2 = getelementptr %IntList, %IntList* %1, i32 0, i32 0
  store i32 1, i32* %2, align 4
  %3 = getelementptr %IntList, %IntList* %1, i32 0, i32 1
  store %IntList* null, %IntList** %3, align 8
  %4 = alloca { i32, %IntList* }, align 8
  %5 = getelementptr { i32, %IntList* }, { i32, %IntList* }* %4,
↪  i32 0, i32 0
  store volatile i32 1, i32* %5, align 4
  %6 = getelementptr { i32, %IntList* }, { i32, %IntList* }* %4,
↪  i32 0, i32 1
  store volatile %IntList* %1, %IntList** %6, align 8
  ret i32 0
}
```

4.2.5. Listas

Al igual que las tuplas, se utilizan los struct de LLVM para representar listas, en este caso con una definición de tipo recursiva.

La implementación de las listas se basa en la idea de una lista encadenada, en la cual la lista se define como un conjunto de nodos enlazados, siendo cada nodo una estructura que contiene el valor del nodo y el puntero al siguiente nodo. La ventaja de este tipo de implementación frente a otras como por ejemplo un arreglo estático, es que se hace mejor uso de la memoria.

Las listas se generan dinámicamente en tiempo de ejecución, y por ende sus nodos deben ser alojados en el heap en lugar del stack como se hace con el resto de las variables. Esto además permite que las funciones devuelvan listas como resultado, dado que si se almacenaran en el stack, la referencia se pierde al volver de la función. Como es usual en este tipo de estructuras de datos, las listas son accesibles mediante un puntero al primer elemento. Para recorrer la lista, se debe seguir los punteros de cada nodo hasta llegar al nodo deseado.

Para construir cada nodo se hace uso de la función `malloc` de C, con esta función se puede alojar memoria para nodos de enteros, punto flotante o booleanos. Esto es diferente a cómo se aloja memoria para los tipos de datos atómicos utilizando la función `alloca` nativa de LLVM, que ocupa memoria del stack (LLVM, 2024).

A nivel de LLVM IR, se definen los tipos `IntList`, `FloatList` y `BoolList` como tipos base en el entorno de ejecución; internamente, estos tipos son una estructura autoreferenciada que define un puntero a un nodo:

```
%IntList = type { i32, %IntList* }
%BoolList = type { i1, %BoolList* }
%FloatList = type { double, %FloatList* }
```

Las tres mencionadas anteriormente son definiciones de tipo con nombre que se usan internamente en LambdaKal, pero el programador utiliza los tipos `[int]`, `[double]` o `[bool]` para referirse a una lista. Estas anotaciones de tipos son traducidas por el módulo generador de código de LambdaKal a su nombre correspondiente en IR.

4.2.6. Let

No existe un equivalente a *let* en LLVM IR, por lo cual las variables declaradas dentro de un *let* simplemente se generan como variables locales previas al bloque dentro del *let*. El espacio para estas variables es reservado mediante la instruccion `alloca` de LLVM IR. Esto implica que las variables solo existen en el entorno local de la función donde se declare el `let`, o en `main` si se declara en *top level*.

Las variables declaradas en la construcción `let` cuentan con inferencia de tipos. Estos tipos se infieren a partir de los componentes del AST de LambdaKal. Esto es posible ya que cada subexpresión de LambdaKal tiene un tipo bien definido embebido en el AST generado.

```
; let a = 1, b = 2 in a + b;
%1 = alloca i32, align 4
store i32 1, i32* %1, align 4
%2 = load i32, i32* %1, align 4

%3 = alloca i32, align 4
store i32 2, i32* %3, align 4
%4 = load i32, i32* %3, align 4

%5 = add i32 %2, %4
```

4.2.7. If/else

La implementación de la estructura if/else en LLVM IR se realiza a través de `label` y las instrucciones `phi` y `br` (LLVM, 2024). Dado que LLVM IR está representado en formato SSA (Appel, 1998), no puede existir en el código una variable que tome dos valores diferentes, incluso si las asignaciones de estos valores tienen lugar en flujos diferentes del código. Por lo tanto, la idea de asignar un valor u otro dependiendo de la etiqueta actual no es posible.

En LLVM IR, se pueden definir bloques de código mediante etiquetas (label). La instrucción `br` permite que la ejecución del código continué a partir de un bloque especifico, similar a como se hace en un lenguaje ensamblador.

La forma canónica de implementar instrucciones de control de flujo es mediante el uso de la instrucción `phi` (Rodler y Egevig, s.f.). Esta instrucción impone una manera de resolver los `if` de manera inversa a la usual: lo que hace la instrucción `phi` es devolver uno de dos resultados dependiendo del bloque que se haya ejecutado previo a la instrucción.

En la Figura 4.12 se observa en primer lugar, una instrucción `br` que continúa la ejecución en la etiqueta `%if.then_0` o `%if.else_0` dependiendo de una condición (que en este caso es la constante `true` para simplificar). Dentro de cada una de las dos etiquetas mencionadas anteriormente, existe una instrucción `br` que dirige al bloque final. En el bloque final, identificado por `%if.end_0`, se invoca a la instrucción `phi` para determinar el valor final del `if`: Si la etiqueta anterior en el flujo de ejecución fue `%if.if_exit_0` entonces se toma el valor 10, mientras que si la etiqueta anterior fue `%if.else_exit_0` se toma el valor 20.

```
; if true then 10 else 20;
br i1 true, label %if.then_0, label %if.else_0

if.then_0:                        ; preds = %0
  br label %if.if_exit_0

if.else_0:                        ; preds = %0
  br label %if.else_exit_0

if.if_exit_0:                     ; preds = %if.then_0
  br label %if.end_0

if.else_exit_0:                   ; preds = %if.else_0
  br label %if.end_0

if.end_0:                         ; preds = %if.else_exit_0,
↪  %if.if_exit_0
  %1 = phi i32 [ 10, %if.if_exit_0 ], [ 20, %if.else_exit_0 ]
```

Figura 4.12: representación de If then else de LambdaKal en LLVM IR

Como observación, esta estructura se puede simplificar, bastaría con tres etiquetas en lugar de cinco, dado que las etiquetas `if.if_exit_0` y `if.else_exit_0` son tan solo pasos intermedios para llegar a `if.end_0`, sin hacer ninguna operación. El motivo para insertar estas dos etiquetas intermedias surge de la necesidad de permitir instrucciones if anidadas. Uno de los ejemplos mas simples es la función `all` de la biblioteca estándar.

```
def all([int] l, fun (int) -> bool f) -> bool:
  if l == [] then
    true
  else
    (if f(head l) then
      all(tail l, f)
    else
      false);
```

Lo que sucede en este caso es que la/s instrucción/es if internas generan etiquetas intermedias, que no se tienen en cuenta dentro de la instrucción phi de `if.end_0`. Entonces, existen dos soluciones, la primera es mantener en un estado cuál fue la última etiqueta en la que termina cada lado del if else, y utilizar estas etiquetas en la instrucción phi. La segunda opción, que es la más sencilla y la que se optó por implementar, es unificar el flujo de cada rama del if

else, mediante la inserción de las dos etiquetas intermedias mencionadas. Con el uso de estas etiquetas, se garantiza que la etiqueta `if.end` siempre es precedida por las etiquetas `if.if_exit` y `if.else_exit`, de esta forma la instrucción phi se puede generar sin tener en cuenta el flujo de los if anidados. En la sección B de anexo se explica más en detalle este razonamiento.

4.2.8. Funciones de alto orden

Las funciones de alto orden se implementan haciendo uso del concepto de "puntero a función" de LLVM. Cuando el programador define una función que recibe una función como parámetro, se genera una definición de función en IR que recibe un puntero a una función; dicho puntero se provee a la función únicamente en el momento de invocarla. De esta forma, la función de alto orden puede desreferenciar el puntero e invocar a la función provista. A diferencia de otras llamadas a función, cuando se tiene una función en el contexto de otra de alto orden, la invocación no está ligada a ningún nombre, ya que el mismo se provee al momento de llamar a la función de alto orden.

```
define i32 @foo(i32 (i32)* %g_0) {
  %1 = call i32 %g_0(i32 9)
  ret i32 %1
}

define i32 @bar(i32 %x_0) {
  %1 = mul i32 %x_0, 3
  ret i32 %1
}

define i32 @main() {
  %1 = call i32 @foo(i32 (i32)* @bar)
  %2 = call i32 @printi(i32 %1)
  ret i32 %2
}
```

En este ejemplo, la función `foo` recibe como parámetro un puntero a una función `g` e invoca esta función con la constante 9. La función 'bar' es definida globalmente para luego invocar a `foo(bar)`.

4.2.9. Funciones externas

La declaración de funciones externas se implementa en LLVM IR mediante la instrucción `declare`, que permite declarar definiciones de funciones sin cuerpo. La traducción es bastante sencilla, solo es necesario proveer el nombre, los parámetros y el tipo de retorno de la función.

```
; extern printi(int i) -> int;
declare i32 @printi(i32)
```

Una vez declarada una función externa, se puede invocar si la misma está definida a través del enlazado dinámico de una biblioteca.

4.3. Optimización

Una de las principales razones para utilizar LLVM como Backend es que provee herramientas de optimización de código. Estas herramientas son particularmente interesantes para optimizar lenguajes funcionales y con intérprete como es el caso de LambdaKal. No solo cuenta con optimizaciones útiles para programación funcional como *tail call elimination*, sino que también puede hacer estas optimizaciones en tiempo de ejecución gracias al JIT. Esto mejora la performance de LambdaKal tanto en modo compilado como en modo intérprete.

La diferencia más grande se ve cuando se pasa del nivel de optimización 1 al 2. Esto tiene sentido ya que como se puede ver en la Figura 4.13, ocurre una llamada recursiva a la función `divisibleAny`, mientras que en la Figura 4.14 esta llamada recursiva se elimina y se usa una iteración en su lugar.

```
1  define i1 @divisibleAny(i32 %n_0, i32 %i_0) local_unnamed_addr
   ↪  #1 {
2    %1 = icmp ult i32 %i_0, 2
3    br i1 %1, label %if.end_1, label %if.else_0
4
5  if.else_0:                                        ; preds = %0
6    %2 = urem i32 %n_0, %i_0
7    %3 = icmp eq i32 %2, 0
8    br i1 %3, label %if.end_1, label %if.else_1
9
10 if.else_1:                                        ; preds =
   ↪  %if.else_0
11   %4 = add i32 %i_0, -1
12   %5 = call i1 @divisibleAny(i32 %n_0, i32 %4)
13   br label %if.end_1
14
15 if.end_1:                                         ; preds =
   ↪  %if.else_0, %if.else_1, %0
16   %6 = phi i1 [ false, %0 ], [ %5, %if.else_1 ], [ true,
   ↪  %if.else_0 ]
17   ret i1 %6
18 }
```

Figura 4.13: Código LLVM IR para encontrar divisores de un número, con optimización nivel 1. Existe una llamada recursiva a `divisibleAny` (línea 12)

```
define i1 @divisibleAny(i32 %n_0, i32 %i_0) local_unnamed_addr #1 {
  %1 = icmp ult i32 %i_0, 2
  br i1 %1, label %if.end_1, label %if.else_0

if.else_0:                                        ; preds = %0, %if.else_0
  %i_0.tr1 = phi i32 [ %4, %if.else_0 ], [ %i_0, %0 ]
  %2 = urem i32 %n_0, %i_0.tr1
  %3 = icmp eq i32 %2, 0
  %4 = add i32 %i_0.tr1, -1
  %5 = icmp ult i32 %4, 2
  %or.cond = or i1 %3, %5
  br i1 %or.cond, label %if.end_1, label %if.else_0

if.end_1:                                         ; preds = %if.else_0, %0
  %6 = phi i1 [ false, %0 ], [ %3, %if.else_0 ]
  ret i1 %6
}
```

Figura 4.14: Código LLVM IR para encontrar divisores de un número, con optimización nivel 2. La iteración sucede en el bloque `if.else_0`

Capítulo 5

Experimentación

5.1. Performance

En esta sección se analizará la eficacia de las optimizaciones de LLVM mediante evidencia empírica. Para ello, se diseñaron una serie de pruebas que permiten observar el impacto del nivel de optimización en el rendimiento de un programa arbitrario escrito en LambdaKal. También se compararán los resultados obtenidos con un programa escrito en Haskell con GHC y GHCi 8.10.7, con las optimizaciones por defecto como punto de referencia.

Los tiempos fueron medidos con el comando `time` en una maquina con un procesador Apple M3 y 16GB de memoria. Los comandos y programas utilizados se encuentran en el repositorio (LambdaKal, 2025) bajo el directorio `examples/bench`.

5.1.1. Pruebas de performance en suma

La primera prueba consiste de un programa sencillo que computa la suma de números del 1 al N, mostrado a continuación.

```
def sum([double] l) -> double:
    if l == [] then 0.0 else (head l) + sum(tail(l));

def one_to_n(double n) -> [double]:
    if n <= 0.0 then [] else n : one_to_n(n - 1.0);

sum(one_to_n(10000.0));
```

Dado que la implementación es recursiva, este programa está limitado por el tamaño del stack. En las pruebas realizadas, se puede ver que a partir de cierto N el programa falla debido a un *Stack Overflow*, indicado en el Cuadro 5.1 mediante *SO*. Al evaluar un programa equivalente en Haskell, también llega a un fallo en la ejecución, aunque a partir de un N mayor.

Compilador	N=1000	N=10k	N=100k	N=1m	N=10m	N=100m
LambdaKal o0	0.104s	0.148s	0.147s	SO	SO	SO
LambdaKal o3	0.147s	0.134s	0.134s	SO	SO	SO
LambdaKal o3 -c	0.001s	0.001s	0.005s	SO	SO	SO
Haskell (ghci)	0.120s	0.110s	0.124s	0.361s	2.825s	SO
Haskell (ghc)	0.002s	0.002s	0.007s	0.065s	0.715s	SO

Cuadro 5.1: Comparación de distintos niveles de optimización en el algoritmo `sum`

Estos resultados pueden ser mejorados utilizando *tail call elimination* (Steele, 1977). En LambdaKal, es necesario cambiar la estructura de la función para que pase a ser una recursión de cola. Con este cambio, LLVM puede implementar la optimización de tail call, si se usa un nivel de optimización o2 o superior.

Al agregar un acumulador a las funciones, se puede utilizar recursión de cola en las funciones `one_to_n_acc` y `sum_acc`.

```
def sum_acc([double] l, double acc) -> double:
  if l == [] then acc else sum_acc(tail l, acc + (head l));

def one_to_n_acc(double n, [double] acc) -> [double]:
  if n <= 0.0 then acc else one_to_n_acc(n - 1.0, n : acc);

sum_acc(one_to_n_acc(1000000.0, []), 0.0);
```

Compilador	N=1000	N=10k	N=100k	N=1m	N=10m	N=100m
LambdaKal o0	0.115s	0.114s	0.124s	SO	SO	SO
LambdaKal o3	0.113s	0.134s	0.135s	0.144s	0.314s	3.527s
LambdaKal o3 -c	0.001s	0.001s	0.003s	0.028s	0.220s	1.843s
Haskell (ghci)	0.125s	0.102s	0.118s	0.510s	4.259s	14.712s
Haskell (ghc)	0.002s	0.002s	0.003s	0.014s	0.129s	1.299s

Cuadro 5.2: Comparación de distintos niveles de optimización en el algoritmo `sum`, con tail call elimination.

Con este cambio, LambdaKal logra ejecutar con valores de N mas altos cuando se corre con optimizaciones de nivel tres habilitadas (`o3`).

Por otro lado, la implementación en Haskell se cambió para usar una implementación con acumulador y recursión de cola, y con evaluación estricta. Esta implementación obtuvo mejores resultados como se puede ver en el Cuadro 5.2.

Observando el código LLVM IR generado por LambdaKal para la función con acumulador y recursión de cola, se puede apreciar que no existen llamadas recursivas en el bloque de la función, sino que ocurre una iteración sobre la etiqueta `if.else_0`.

```
define double @sum_acc.31(%FloatList* %l_0, double %acc_0)
↪  local_unnamed_addr #2 {
  %1 = ptrtoint %FloatList* %l_0 to i64
  %2 = trunc i64 %1 to i32
  %3 = icmp eq i32 %2, 0
  br i1 %3, label %if.end_0, label %if.else_0

if.else_0:                                ; preds = %0, %if.else_0
  %acc_0.tr2 = phi double [ %8, %if.else_0 ], [ %acc_0, %0 ]
  %l_0.tr1 = phi %FloatList* [ %5, %if.else_0 ], [ %l_0, %0 ]
  %4 = getelementptr %FloatList, %FloatList* %l_0.tr1, i64 0,
↪  i32 1
  %5 = load %FloatList*, %FloatList** %4, align 8
  %6 = getelementptr %FloatList, %FloatList* %l_0.tr1, i64 0,
↪  i32 0
  %7 = load double, double* %6, align 8
  %8 = fadd double %acc_0.tr2, %7
  %9 = ptrtoint %FloatList* %5 to i64
  %10 = trunc i64 %9 to i32
  %11 = icmp eq i32 %10, 0
  br i1 %11, label %if.end_0, label %if.else_0

if.end_0:                                 ; preds = %if.else_0, %0
  %acc_0.tr.lcssa = phi double [ %acc_0, %0 ], [ %8, %if.else_0
↪  ]
  ret double %acc_0.tr.lcssa
}
```

5.1.2. Pruebas de performance en *Selection sort*

Otra de las pruebas realizadas para evaluar la eficacia de la optimización de LLVM en LambdaKal consiste en ejecutar el algoritmo *selection sort* sobre listas aleatorias de largo N. Se eligió este algoritmo ya que es un problema bien estudiado para el cual existen optimizaciones. Como escala de forma no lineal con la entrada, se puede observar mejor el impacto de las optimizaciones. Se hicieron pruebas en intervalos de 1000 elementos, desde N=1000 hasta N=5000, donde en cada intervalo se mide el tiempo de ejecución de las siguientes implementaciones:

1. LambdaKal sin optimización (argumento `-o0`)

2. LambdaKal con optimización máxima (argumento `-o3`)

3. Programa equivalente interpretado en Haskell con GHCi 8.10.7

La implementación de *Selection sort* utilizada es mostrada a continuación. Esta función y las funciones llamadas dentro de la misma forman parte de biblioteca estándar de LambdaKal.

```
def selection_sort_int([int] arr) -> [int]:
  if arr == [] then
    []
  else
    let min_tuple = find_min_int(arr, 9999999) in
    let min_val = fst(min_tuple) in
    let min_idx = snd(min_tuple) in
    let rest = remove_nth_int(arr, min_idx) in
    min_val : selection_sort_int(rest);
```

En la Tabla 5.3 se muestran los tiempos promedio de ejecución de cada variante. Observar que la optimización de LLVM tiene mayor efecto para números grandes, mientras que para números pequeños el efecto es menor o incluso negativo ya que el proceso de optimización del JIT ocurre durante la ejecución del programa. En la Figura 5.1 se puede observar claramente las diferencias entre la versión optimizada y no optimizada.

Compilador	N=1000	N=2000	N=3000	N=4000	N=5000
LambdaKal o0	0.157s	0.206s	0.254s	0.378s	0.538s
LambdaKal o3	0.170s	0.168s	0.225s	0.316s	0.407s
Haskell (ghci)	0.188s	0.207s	0.263s	0.347s	0.413s

Cuadro 5.3: Comparación de distintos niveles de optimización en el algoritmo *Selection sort*

Para observar el efecto de la optimización únicamente sobre la ejecución del programa, sin tener en cuenta el tiempo consumido por los ciclos de optimización en sí, se repitieron las pruebas ejecutando el código precompilado. Para esto se hizo uso de la funcionalidad de compilación de LambdaKal mediante su argumento de consola `-c`. Para poder realizar este experimento de forma objetiva, fue necesario modificar la parte del código que invoca a *Clang*, para que omita las optimizaciones de dicho compilador, conservando únicamente las optimizaciones que hace LLVM. En el Cuadro 5.4 se ven los tiempos de ejecución sobre los respectivos archivos ejecutables

Compilador	N=1000	N=2000	N=3000	N=4000	N=5000
LambdaKal o0	0.014s	0.052s	0.169s	0.205s	0.322s
LambdaKal o3	0.011s	0.046s	0.094s	0.170s	0.240s
Haskell (ghc)	0.008s	0.024s	0.049s	0.086s	0.133s

Cuadro 5.4: Comparación de distintos niveles de optimización en el algoritmo *Selection sort*, corriendo sobre el archivo ejecutable precompilado.

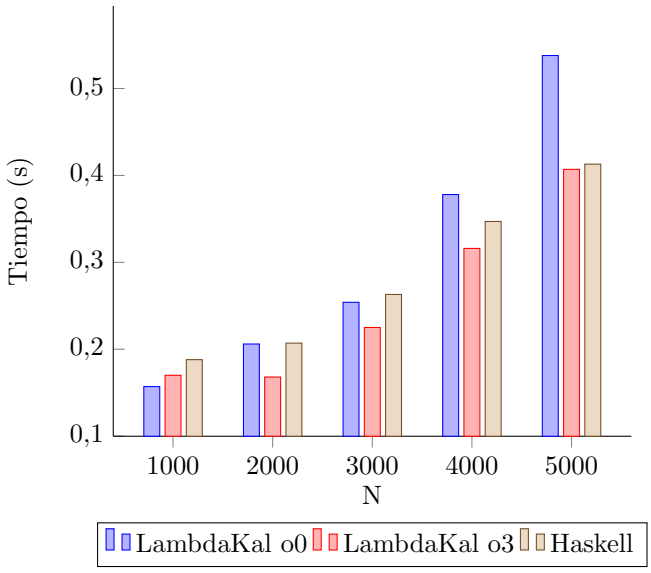

Figura 5.1: Comparación de tiempo de ejecución de un algoritmo *Selection sort* en: LambdaKal sin optimización, LambdaKal con optimización nivel 3 y Haskell

En la Figura 5.2, se puede observar que el tiempo de ejecución en general es mucho menor, y que en todos los casos el código optimizado fue más rápido que su contraparte no optimizada. Haskell compilado con GHC resultó ser aún más rápido, a continuación entraremos mas en detalle.

Por trabajarse en el paradigma funcional se van a crear nodos nuevos de la lista todo el tiempo, cada vez que se hace una modificación. No ocurre lo que pasa en el paradigma imperativo en que se actualizan los nodos de la misma lista cambiando los punteros. Dado que LambdaKal no cuenta con un *garbage collector*, el uso de memoria se dispara, como se puede ver en el Cuadro 5.5. La herramienta usada para medir el uso máximo de memoria fue `/usr/bin/time -v`.

Compilador	N=1000	N=2000	N=3000	N=4000	N=5000
LambdaKal o0	9.26MB	33.088MB	72.644MB	127.464MB	198.844MB
LambdaKal o3	9.256MB	32.956MB	72.74MB	127.168MB	198.8MB
Haskell (ghc)	4.104MB	4.264MB	4.392MB	4.856MB	5.352MB

Cuadro 5.5: Comparación de uso máximo de memoria para distintos niveles de optimización en el algoritmo *Selection sort*, corriendo sobre el archivo ejecutable precompilado

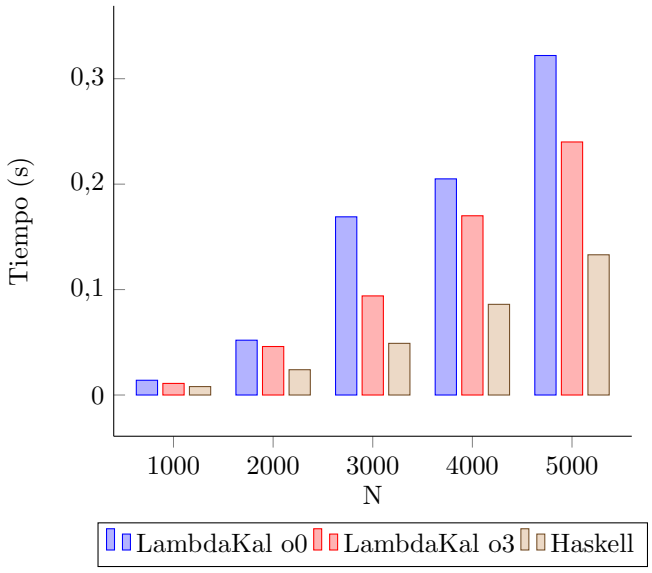

Figura 5.2: Comparación de tiempo de ejecución de un algoritmo *Selection sort* implementado en LambdaKal sin optimización y con optimización nivel tres, desde un archivo ejecutable.

5.1.3. Pruebas de performance en Fibonacci recursivo

En esta serie de pruebas se evalúa como maneja el optimizador funciones que tienen una alta complejidad computacional, como es el caso de Fibonacci recursivo. Se usó la misma función de Fibonacci del ejemplo del Capítulo 3 3.1, que tiene una complejidad computacional $O(2^n)$. También se implementó un programa equivalente en Haskell para usar de referencia, mostrado en la Figura 5.3.

```
module Fib where

fib :: Int -> Int
fib n = if n < 3 then 1 else fib (n - 1) + fib (n - 2)

main :: IO ()
main = print $ fib 35
```

Figura 5.3: Función en Haskell para calcular Fibonacci de forma recursiva.

Para probar, se utilizaron valores de N entre 35 y 40, dado que valores más altos tienen un tiempo de ejecución muy alto.

Compilador	N=36	N=37	N=38	N=39	N=40
LambdaKal o0	0.133s	0.155s	0.203s	0.263s	0.364s
LambdaKal o3	0.167s	0.143s	0.184s	0.224s	0.304s
Haskell (ghc)	0.339s	0.591s	0.884s	1.429s	2.350s
Haskell (ghci)	5.043s	8.016s	13.074s	21.160s	34.486s

Cuadro 5.6: Tiempos de ejecución del algoritmo Fibonacci, graficados en la Figura 5.4

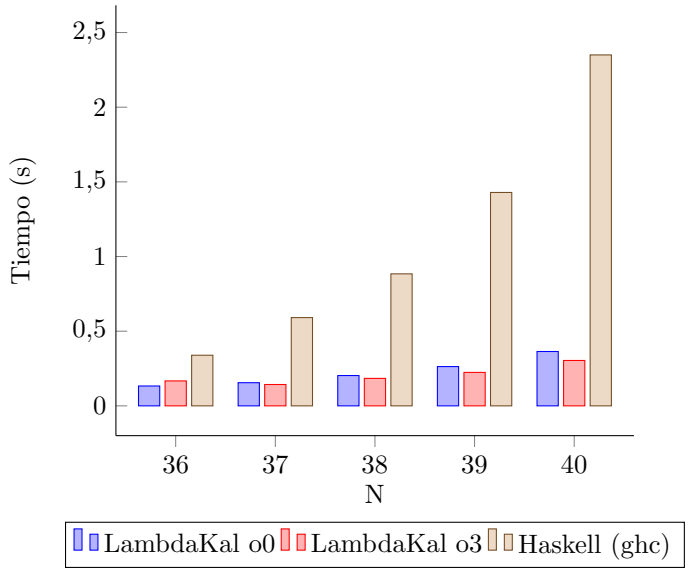

Figura 5.4: Comparación de tiempo de ejecución de un algoritmo *Fibonacci* implementado en LambdaKal sin optimización, con optimización nivel tres y Haskell compilado con GHC

Estos resultados muestran que LLVM, al igual que Haskell, no puede optimizar cualquier algoritmo si la estructura y lógica del mismo no están expresadas de forma apropiada. Analizando la salida del código LLVM IR, se puede ver que se hacen algunas optimizaciones como, por ejemplo, transformar una de las llamadas recursivas en una iteración. Sin embargo, analizando el código generado, el algoritmo sigue teniendo la misma complejidad computacional de $O(2^n)$.

En el Cuadro 5.7 se puede observar que el uso máximo de memoria no tiende a crecer con N como sí lo hace en *selection sort*. Como el uso de memoria en este algoritmo no es alto, la falta de *garbage collector* en LambdaKal no es un problema tan grave. Esto puede explicar por qué en este caso LambdaKal tiene una performance mejor.

Observando el código LLVM generado con optimización nivel tres, se encuentra que parte de la función recursiva ha sido transformada a un *loop* (iterando

Compilador	N=36	N=37	N=38	N=39	N=40
LambdaKal o0	1056KB	1156KB	1016KB	1048KB	1044KB
LambdaKal o3	1068KB	1020KB	1020KB	1052KB	1044KB
Haskell (ghci)	166696KB	166684KB	166896KB	166772KB	167136KB
Haskell (ghc)	3988KB	3848KB	3848KB	4016KB	4520KB

Cuadro 5.7: Comparación de uso máximo de memoria para distintos niveles de optimización en el algoritmo Fib

en la etiqueta de la línea 6). Sin embargo, en este *loop* se hace una llamada recursiva en la línea 10. Curiosamente, en la línea 11 se puede ver que el número utilizado como argumento de la llamada recursiva se le resta dos, es decir, que la iteración ocurre de a dos pasos, salteándose una llamada en el medio. Esto sigue siendo un algoritmo de $O(2^n)$ pero en la práctica logra tiempos ligeramente mejores, como se puede ver en el Cuadro 5.6.

```
1  ; Function Attrs: nounwind readnone
2  define i32 @fib.16(i32 %x_0) local_unnamed_addr #4 {
3    %1 = icmp slt i32 %x_0, 3
4    br i1 %1, label %if.end_0, label %if.else_0
5
6  if.else_0:                                        ; preds = %0,
   ↪  %if.else_0
7    %x_0.tr2 = phi i32 [ %4, %if.else_0 ], [ %x_0, %0 ]
8    %accumulator.tr1 = phi i32 [ %5, %if.else_0 ], [ 0, %0 ]
9    %2 = add nsw i32 %x_0.tr2, -1
10   %3 = tail call i32 @fib.16(i32 %2)
11   %4 = add nsw i32 %x_0.tr2, -2
12   %5 = add i32 %3, %accumulator.tr1
13   %6 = icmp slt i32 %x_0.tr2, 5
14   br i1 %6, label %if.end_0.loopexit, label %if.else_0
15
16 if.end_0.loopexit:                                ; preds =
   ↪  %if.else_0
17   %phi.bo = add i32 %5, 1
18   br label %if.end_0
19
20 if.end_0:                                         ; preds =
   ↪  %if.end_0.loopexit, %0
21   %accumulator.tr.lcssa = phi i32 [ 1, %0 ], [ %phi.bo,
   ↪  %if.end_0.loopexit ]
22   ret i32 %accumulator.tr.lcssa
23 }
```

En otras palabras, la lógica de la función fue modificada a la siguiente forma,

donde en lugar de dos llamadas recursivas, pasa a ser un *loop* con una llamada recursiva.

```
1: function FIB(x)
2:     if x < 3 then
3:         return 1
4:     end if
5:     accumulator ← 0
6:     current_x ← x
7:     while true do
8:         result ← FIB(current_x − 1)
9:         accumulator ← accumulator + result
10:        current_x ← current_x − 2
11:        if current_x < 5 then
12:            return accumulator + 1
13:        end if
14:    end while
15: end function
```

5.2. Testing

Para mejorar el proceso de mantenimiento y verificación del proyecto, se implementaron desde el principio pruebas de regresión (Naik y Tripathy, 2011). Estas pruebas consisten en proveer al intérprete con diversos programas escritos en LambdaKal, y evaluar su resultado contra un resultado esperado.

La suite de pruebas es fácilmente extensible, se pueden crear nuevos test simplemente agregando un archivo con extensión .k y su correspondiente resultado esperado en formato de texto en otro archivo. El directorio para programas de prueba es `/test/programs` y para los resultados esperados `/test/output`. En caso de no existir un archivo con el resultado esperado, este se crea automáticamente con contenido igual a la salida de la ejecución del programa de entrada.

La suite de tests recorre todos los archivos de código fuente en el directorio de tests y los ejecuta mediante una invocación del ejecutable del lenguaje. Esto significa que estos tests se comportan como pruebas de "caja negra": cada caso de prueba simplemente obtiene una entrada y compara su salida con un valor esperado. Al ser ejecutados de esta forma, cada caso de prueba verifica el flujo entero del intérprete, incluyendo el *parsing* de las instrucciones, la generación de código y la impresión en pantalla de los resultados.

En esta versión se tienen 101 archivos de código LambdaKal que se usan como entrada para la suite de tests. Cada test evalúa cada archivo dos veces; la primera sin optimización y la segunda con optimización máxima. Esto da un total de 202 tests individuales.

En general, los tests están enfocados a una funcionalidad específica del lenguaje. Por ejemplo, uno de los primeros tests implementados es un programa que suma un número entero con un número en punto flotante (ver Figura 5.5).

Esta prueba sirve para comprobar el correcto funcionamiento de la parte del generador de código que genera instrucciones de suma (`add` o `fadd`) según el tipo del operando.

```
-- test/programs/add_int_float.k
1 + 1.0;

-- test/output/add_int_float.k
2.000000
```

Figura 5.5: Programa de entrada y archivo de salida esperada del primer test

Al tener este test, se puede comprobar que las siguientes versiones del lenguaje conserven el comportamiento esperado para la suma entre un entero y un número en punto flotante.

Muchos de los tests fueron creados a partir de problemas encontrados en ciertas versiones del lenguaje, por ejemplo, la funcionalidad de listas pasó por varias implementaciones, y en alguna de ellas había problemas al devolver una lista desde una función. A raíz de esto, se creó un caso de prueba para comprobar que las funciones pudieran devolver una lista correctamente.

```
-- test/programs/fn_create_list_float.k
def numbers() -> [double]: [2.81, 3.14, 4.0];

numbers();

-- test/output/fn_create_list_float.k
[2.810000, 3.140000, 4.000000]
```

Los tests resultaron de mucha utilidad al momento de agregar nuevas funcionalidades al lenguaje, ya que se pudo continuar con la implementación de las mismas teniendo la seguridad de poder detectar si se deshacen accidentalmente otras funcionalidades del lenguaje.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

En este trabajo se logró desarrollar un compilador y un intérprete de lenguaje puramente funcional (llamado LambdaKal), usando Haskell y la infraestructura de compilación de LLVM. El lenguaje LambdaKal cuenta con características fundamentales como estructuras de datos (listas y tuplas), definición de funciones, funciones recursivas y de alto orden, ofreciendo una base sólida para implementar programas simples.

Durante el desarrollo se implementaron funcionalidades que van más allá de lo provisto por el tutorial de referencia Kaleidoscope, lo que requirió una investigación profunda sobre el funcionamiento interno de LLVM, sus primitivas de bajo nivel, y los *bindings* entre Haskell y LLVM. Esta exploración permitió adquirir y documentar conocimientos acerca de la generación de código intermedio (LLVM IR) y su integración con Haskell.

Una de las funcionalidades más destacadas fue la implementación de un entorno interactivo REPL, que conserva el mismo conjunto de capacidades que la ejecución de archivos fuente. Esto fue posible gracias a la capacidad de compilación JIT provista por LLVM.

Si bien el lenguaje cumple con los objetivos establecidos, existen múltiples direcciones para su expansión futura. Entre ellas se destacan la incorporación de nuevos tipos de datos como *strings*, funciones anónimas y listas anidadas. Ninguna de estas características requiere cambios radicales, pero quedaron fuera del alcance de este proyecto por razones de tiempo.

El lenguaje tiene un rendimiento eficiente, aunque no incorpora mejoras a nivel de lenguaje como por ejemplo *lazy evaluation* o *garbage collection*. Todas las mejoras de rendimiento se logran a través de las optimizaciones aplicadas al código intermedio por parte de LLVM.

La suite de testing desarrollada es amplia, automatizada y fácil de mante-

ner, lo cual garantiza la estabilidad del lenguaje frente a cambios futuros. Esta herramienta fue fundamental para validar el comportamiento del compilador en sus distintas fases y para sostener una arquitectura modular y extensible.

En conclusión, este proyecto muestra la viabilidad de implementar un lenguaje funcional moderno utilizando Haskell y LLVM. Es un caso práctico que sienta las bases para futuros trabajos de implementación de lenguajes sobre la infraestructura LLVM.

6.2. Trabajo futuro

A continuación se listan algunas de las características que se podrían agregar a futuro sobre la base de LambdaKal

6.2.1. Lazy evaluation

Una característica interesante a implementar sería *lazy evaluation* para las funciones y permitir trabajar con estructuras infinitas.

```
ready> def inf_list() -> [int]: 1:inf_list();
ready> head(inf_list());
```

En LambdaKal, esta expresión resulta en un error en tiempo de ejecución dado que se genera una recursión infinita. Si se tuviera *lazy evaluation*, no se intentaría computar toda la lista, y el resultado de esta expresión sería `1`. *Lazy evaluation* no está disponible de forma nativa en LLVM, por lo tanto, sería necesario implementar toda esta lógica previo a la generación del código IR. Esto supone un gran desafío desde un punto de vista técnico y requiere una reestructuración completa de la lógica de producción y evaluación de código.

6.2.2. Pattern matching

Pattern matching es la capacidad de comparar una secuencia de tokens para encontrar los componentes de algún patrón. Esto es una característica especialmente útil en el paradigma de programación funcional, dado que se pueden dar distintas definiciones de una función según las características de los parámetros.

Desde un punto de vista técnico, no sería muy difícil de incorporar al lenguaje, si bien LLVM IR no utiliza *pattern matching*, se puede emular este comportamiento o bien definiendo diferentes versiones de una misma función, o agregando una estructura de control, mediante *labels*, para separar el comportamiento de las funciones según el patrón encontrado.

6.2.3. Funciones de alto orden en retorno

Si bien LambdaKal tiene la capacidad de pasar funciones como parámetro a otras funciones, eso es solo una de las definiciones de alto orden. El otro tipo de

funciones de alto orden son aquellas que devuelven una función como resultado. Con la manera en la que están implementadas las funciones hoy en día, no sería muy difícil agregar esta funcionalidad, ya que el problema de tratar las funciones como punteros ya fue resuelto para pasar funciones como parámetro.

6.2.4. Garbage collection

LLVM no hace *garbage collection*, pero sí provee una interfaz para implementar esto (LLVM, 2025a). Dado que LambdaKal es un lenguaje de alto nivel y que no trabaja con direcciones de memoria directamente, esta característica sería bastante útil, pero no estuvo considerada en el alcance del proyecto.

6.2.5. Polimorfismo

Al momento de definir una función, se debe definir su tipo de forma estricta. No es posible definir una función que tenga tipos genéricos. Esto sería de mucha utilidad, sobre todo para la biblioteca estándar, ya que varias funciones son definidas más de una vez para proveer la función a distintos tipos. Como LLVM usa tipado estático internamente, implementar esta característica no es trivial.

Referencias

Aaron Bloomfield. (2016). *The 64-bit x86 C Calling Convention.* https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf. (Derived from earlier work by Adam Ferrari, Alan Batson, Mike Lack, and Anita Jones)

Adam Paszke. (2021). *Getting to the Point. Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming.* Descargado de https://arxiv.org/abs/2104.05372

Amy Brown and Greg Wilson. (2011). *The Architecture of Open Source Applications*. Lulu.com.

Anastasia Stulova and Sven van Haastregt. (2019, October). *An Overview of Clang.* Arm, Cambridge, UK.

Appel, A. W. (1998, abril). SSA is functional programming. *SIGPLAN Not.*, *33*(4), 17–20.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., y Zadeck, F. K. (1991, octubre). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, *13*(4), 451–490. doi: 10.1145/115372.115320

Free Software Foundation, I. (2025). *GCC Internals Manual.* Descargado de https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

Haskell. (2025a). *llvm-hs.* https://hackage.haskell.org/package/llvm-hs. Autor.

Haskell. (2025b). *llvm-hs-pure.* https://hackage.haskell.org/package/llvm-hs-pure. Autor.

Haskell. (2025c). *Parsec.* https://hackage.haskell.org/package/parsec. Autor.

LambdaKal. (2025). *LambdaKal.* https://github.com/andrescollares/kaleidoscope. GitHub.

Lattner, C., y Adve, V. (2004, Mar). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. En *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04).* Palo Alto, California.

Linux man pages. (2024). *dlsym(3) - Linux man page.* https://linux.die.net/man/3/dlsym. (Accessed: 2024-04-21)

LLVM. (2024). *LLVM Language reference.* https://llvm.org/docs/LangRef.html. (Accessed: 2024-04-21)

LLVM. (2025a). *Garbage Collection with LLVM.* Descargado de https://llvm.org/docs/GarbageCollection.html

LLVM. (2025b). *MCJIT Design and Implementation.* Descargado de https://llvm.org/docs/MCJITDesignAndImplementation.html

LLVM. (2025c). *My First Language Frontend with LLVM Tutorial.* Descargado de https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html

LLVM Tutorials. (2024). *My First Language Frontend with LLVM Tutorial.* https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/

`index.html`. (Accessed: 2024-04-21)

Muchnick, S. S. (1998). *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Naik, K., y Tripathy, P. (2011). *Software Testing and Quality Assurance: Theory and Practice*. Wiley.

Rodler, M., y Egevig, M. (s.f.). *Mapping High Level Constructs to LLVM IR*. Descargado de `https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/`

Simon Thompson. (2011). *Haskell: The Craft of Functional Programming*. Addison Wesley.

S.P. Jones. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press. Descargado de `https://books.google.com.uy/books?id=mMGQgcnCxjAC`

Steele, G. L. (1977). Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. En *Proceedings of the 1977 annual conference* (p. 153–162). New York, NY, USA: Association for Computing Machinery.

Stephen Diehl. (2024). *Implementing a JIT Compiled Language with Haskell and LLVM*. `https://web.archive.org/web/20190329145322/https://www.stephendiehl.com/llvm/`. (Accessed: 2024-04-21)

Wegman, M. N., y Zadeck, F. K. (1991, abril). Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, *13*(2), 181–210. Descargado de `https://doi.org/10.1145/103135.103136` doi: 10.1145/103135.103136

Anexo A

Entorno de desarrollo

El código fuente se encuentra en el repositorio de GitHub (LambdaKal, 2025).

El proyecto está configurado para desarrollar y probar en Docker, por lo cual esta es la única dependencia necesaria para comenzar a desarrollar. Utiliza *Cabal* como sistema de gestión de paquetes de Haskell.

El proyecto se puede iniciar mediante los comandos

```
docker compose build
docker compose up -d
docker compose run project bash
cabal run
```

Cada vez que hay un cambio en el código, se puede recompilar mediante `cabal run`.

En caso de realizar modificaciones en los archivos `C` que constituyen parte de la biblioteca estándar, hay que volver a compilar los *shared object*.

```
cd /lambdakal/src/StdLib/cbits
gcc -fPIC -shared -o /usr/lib/liblambdakal.so io.c list.c
```

Anexo B

Implementación de instrucciones phi

En un if simple, las etiquetas `if.if_exit` y `if.else_exit` no son necesarias. Sin embargo, si se hace eso, el orden de generación de los bloques es:

1. `if.then_0`
2. `if.else_0`
3. `if.then_1`
4. `if.else_0`
5. `if.exit_0`
6. `if.exit_1`

Entonces, `if.exit_1` es precedido por `if.exit_0`, esto es incorrecto, ya que en la condición del `phi`, se asume que va a venir de `if.then` o `if.else`, no de otra etiqueta `if.exit`. Esto produce un error al invocar al intérprete de LLVM.

```
genOperand (S.If cond thenExpr elseExpr) localVars = mdo
  computedCond <- genOperand cond localVars
  condBr computedCond ifThen ifElse
  ifThen <- block `named` "if.then"
  computedThen <- genOperand thenExpr localVars
  br ifExit
  ifElse <- block `named` "if.else"
  computedElse <- genOperand elseExpr localVars
  br elseExit
  ifExit <- block `named` "if.if_exit"
  br blockEnd
  elseExit <- block `named` "if.else_exit"
  br blockEnd
  blockEnd <- block `named` "if.end"
  phi [(computedThen, ifExit), (computedElse, elseExit)]
```

Figura B.1: Función de generación de código de expresiones `if` de LambdaKal

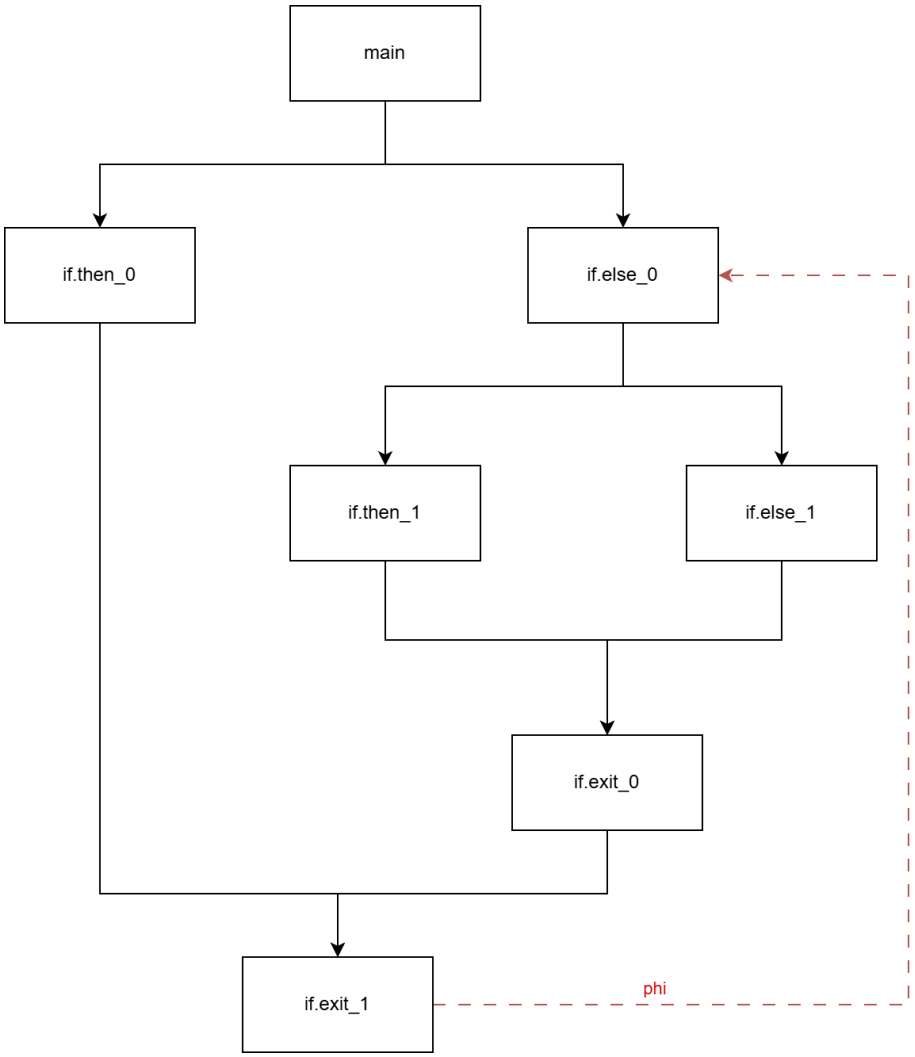

Figura B.2: Implementación de If anidados con tres labels, la label `if.exit_1` es generada asumiendo que la última instrucción va a venir del bloque `if.else_0`, lo cual no sucede si dentro del mismo hay otra instrucción if que genere bloques intermedios.

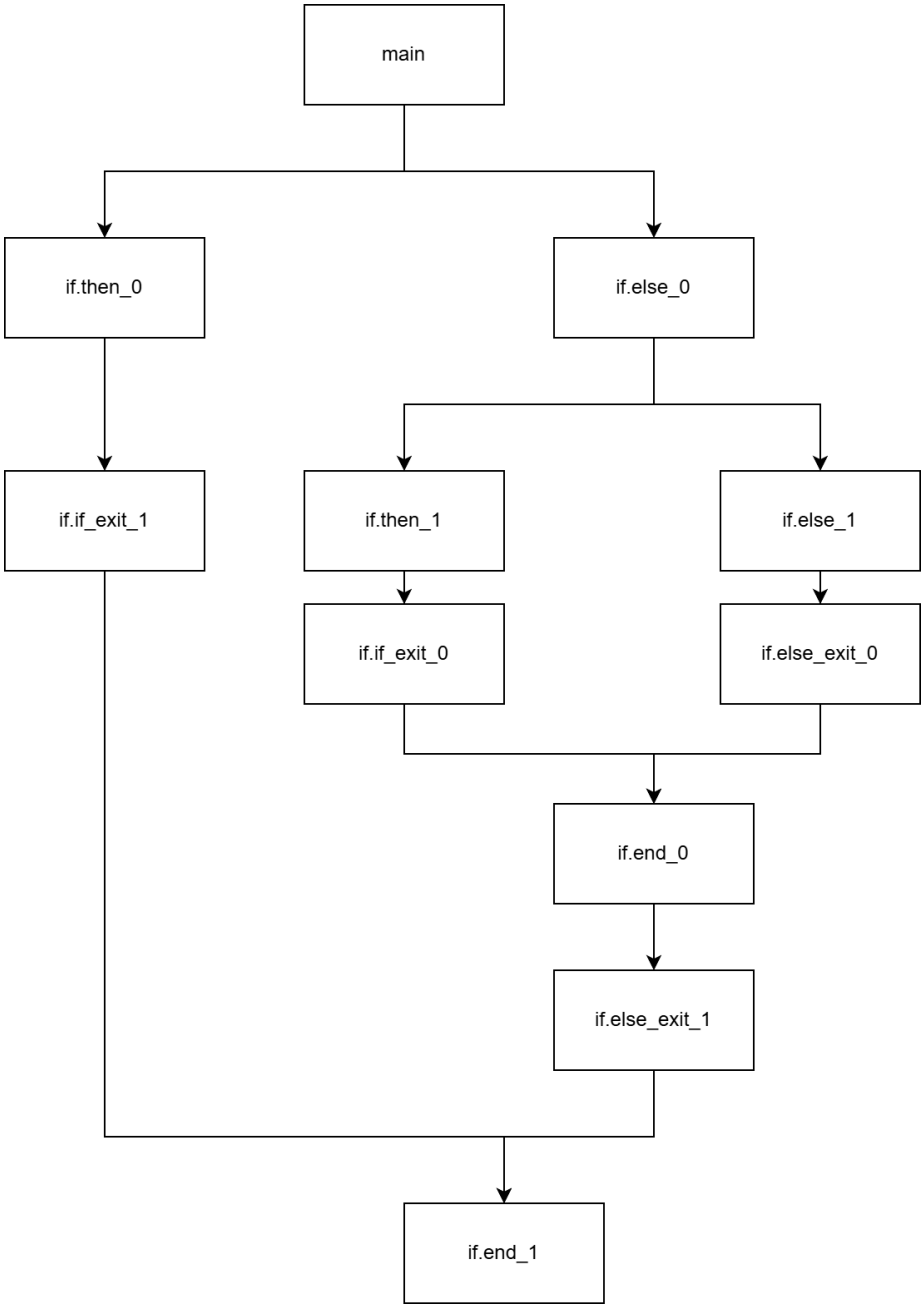

Figura B.3: Implementación de If anidados con 5 labels, las labels `if.if_exit` y `if.else_exit` siempre preceden al bloque `if.end`, por lo cual la instrucción phi puede armarse con estos dos bloques