LLVM Compiler Framework

CI4721 – Lenguajes de Programación II

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

LLVM Compiler System

- **Infraestructura de Compilación LLVM**
  - Componentes reutilizables para construir compiladores.
  - Reduce el tiempo y costo de construir un compilador.
  - Compiladores estáticos, JITs, optimizadores...
- **Entorno de Compilación LLVM**
  - Compiladores *completos* usando LLVM.
  - Front-ends C, C++, Objective C y Fortran – robustos y agresivos.
  - Front-ends Java y Scheme – en progreso muy lento.
  - Emiten C o código nativo X86, MIPS, Alpha, ARM, Sparc o PowerPC.
  - Calidad del código emitido igual o superior a GCC.

«LLVM» quiere decir «LLVM»

UNIVERSIDAD SIMÓN BOLÍVAR

Componentes de LLVM

- **LLVM Virtual Instruction Set**
  - Representación intermedia común independiente del destino.
  - Representaciones interna (IR) y externa (persistente).
- Colección de librerías integradas
  - Análisis para optimización, generación, JIT.
  - Recolección de basura, perfilado.
- Herramientas construidas con las librerías
  - Ensamblador, generador de código, enlazador.
  - Optimizador modular.
  - Debugger.

«LLVM» es para gente que sabe de compiladores –
tanto que no quiere escribir el *backend*

UNIVERSIDAD SIMÓN BOLÍVAR

El compilador C/C++ LLVM

- De “lejos” luce como un compilador estándar
  - Compatible con cualquier *makefile* estándar.
  - Usa el reconocedor C/C++ de GCC 4.2
- De “cerca” aparecen las diferencias.
  - Usa el optimizador LLVM.
  - Los archivos `.o` contienen LLVM IR en lugar de código de máquina.
  - Usa el enlazador LLVM.
  - El ejecutable puede ser bytecode (JIT) o código de máquina.

UNIVERSIDAD SIMÓN BOLÍVAR

Objetivos de LLVM IR

- Fácil de producir, comprender y *definir*.
- Independiente de los lenguajes origen y destino.
- Unica representación para análisis y optimización.
  - Mejoramiento de alto nivel – eliminar código muerto, expresiones comunes, promoción de copias, factorización de lazos, *peephole*...
  - Mejoramiento de bajo nivel – reducción de fuerza aritmética, operaciones escalares, coprocesadores...
- Mejorar tanto como se pueda tan pronto sea posible.
- Facilitar la escritura de nuevos métodos de mejoramiento.

UNIVERSIDAD SIMÓN BOLÍVAR

Instrucciones LLVM IR

- Semántica independiente de la máquina destino.
  - Código de tres direcciones estilo RISC.
  - Conjunto infinito de registros en forma SSA.
  - Construcciones simples para flujo de control.
  - Apuntadores con tipo.
- Tres representaciones
  - Texto – para los humanos.
  - En memoria – para el proceso de compilación.
  - En archivo binario – para integración con otras herramientas.

UNIVERSIDAD SIMÓN BOLÍVAR

Sistema de Tipos de LLVM IR

- Sistema de Tipos
  - Primitivos – etiqueta, void, float, integer.
    - Enteros con cantidad arbitraria de bits (i1, i3, i32, i64).
    - Punto flotante de hasta 128 bits.
  - Agregados – apuntador, arreglo, estructura y función.
  - Vectores – para arquitecturas SIMD.
  - No hay tipos de alto nivel – neutralidad.
- Permite usar *type casts*
  - Para expresar lenguajes con sistemas débiles como C.
  - Implantar sistemas fuertes es problema del *front end*.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo luce LLVM IR?

- TAC – dos registros origen y un registro destino

```
%t2 = add i32 %t0, %t1
```

- TAC – un origen puede ser un valor

```
%t5 = add i32 %t2, 42
```

- Las instrucciones tienen tipo

```
%t8 = fadd double %t6, %t7
store i32 %ts, i32* %r
```

- Cada resultado requiere un nuevo registro – forma *Static Single Assignment* (SSA).

UNIVERSIDAD SIMÓN BOLÍVAR

Hello world!

```
@hw - internal constant [13 x i8] c"hello world\0A\00"
declare i32 @puts(i8*)

define i32 @main () {
entry: %t1 = bitcast [13 x i8]* @hw to i8*
       %t2 = call i32 @puts(i8* %t1)
       ret i32 %t2
}
```

- `@hw` es una constante global.
- Se declara la función externa `@puts` con firma.
- Se hace el *cast* y se llama a la función.

UNIVERSIDAD SIMÓN BOLÍVAR

Instrucciones LLVM IR

- Fácil representación de estructuras de control
  - Saltos – Condicionales tradicionales
  - Saltos indirectos – Calculados (tablas de despacho).
  - Selectores – Específicos para *case/switch*.
  - Excepciones – Selector de *catch* explícito e implícito.
  - Llamadas – Múltiples estilos de secuencia de llamada.
- Información de alto nivel expuesta en el código intermedio.
  - Flujo de datos explícito – por la forma SSA.
  - Grafo de flujo explícito – incluso para excepciones.
  - Información explícita de tipos independiente del lenguaje origen.
  - Aritmética explícita de apuntadores con tipos – preserva la forma de los arreglos.

UNIVERSIDAD SIMÓN BOLÍVAR

Los tipos se hacen concretos en LLVM

- Los tipos abstractos se hacen concretos en LLVM.
  - Tipos complejos en el lenguaje origen se simplifican en LLVM.
  - De implícitos y abstractos pasan a explícitos y concretos.
- Pero no tienes que bajar tanto de nivel, e.g.
  - Referencias (T&) pasan a apuntadores (T*).
  - `complex` pasa a `( float, float )`.
  - *Bitfields* `struct { int foo:4, int bar:2 }` pasa a `{ i6 }`.
  - Métodos como `class Foo { void bar() }`
    pasan a funciones con argumentos implícitos `bar(Foo*)`

UNIVERSIDAD SIMÓN BOLÍVAR

Estructura de un programa en LLVM IR

- **Módulo**
  - Contiene variables globales y funciones.
  - Es la unidad de compilación o análisis.
- **Funciones**
  - Contiene argumentos y bloques básicos.
  - Corresponden a las funciones de C en la práctica.
- **Bloque Básico**
  - Contiene una lista de instrucciones.
  - Termina con una instrucción de salto.
- **Instrucción**
  - Código de operación y vector de operandos.
  - *Todos* los operandos tienen tipo asociado.
  - El resultado de la operación tiene tipo asociado.

Detalles internos y API

- LLVM está escrito en C++ – explota STL (vector, set y map).
- Representación interna de IR simple
  - Listas doblemente enlazadas.
  - Módulo tiene lista de funciones y globales.
  - Función tiene lista de bloques básicos y argumentos.
  - Bloque tiene lista de instrucciones.
- Se usan iteradores para recorrer las listas.

Más fácil de comprender que GCC.

UNIVERSIDAD SIMÓN BOLÍVAR

Organización del compilador LLVM

- Serie de "pasadas" – cada pasada implanta **una** transformación.
- Las pasadas pueden ser:
  - De Módulo – para estudio entre procedimientos.
  - Flujo de llamadas – de abajo hacia arriba.
  - Por función – para analizar una función aislada.
  - Por bloque – para analizar un bloque aislado.
- Restricciones para asegurar aislamiento.
- Escribir una "pasada" se reduce a crear una subclase.

UNIVERSIDAD SIMÓN BOLÍVAR

Las herramientas LLVM

- Primitivas – hacen una sola cosa, pero muy bien.
  - `llvm-as` – de texto IR (`.ll`) a binario IR (`.bc`).
  - `llvm-dis` – de binario IR (`.bc`) a texto IR (`.ll`).
  - `llvm-link` – enlaza múltiples binarios (`.bc`).
  - `llvm-prof` – emite reportes de profiling.
  - `llvmc` – director de compilación configurable.
  - . . . y varias más.
- Agregadas – aprovecha primitivas y otras cosas.
  - `gccas` – optimizador C/C++ a tiempo de compilación.
  - `gccld` – optimizador C/C++ a tiempo de enlazado.
  - `llvm-gcc` y `llvm-g++` – los compiladores C y C++.

UNIVERSIDAD SIMÓN BOLÍVAR

El optimizador es modular

- Posible invocar una secuencia arbitraria de pasadas.
  - El *PassManager* se controla desde la línea de comandos.
  - Permite cargar pasadas como *plug-ins*.
- Cada pasada se "registra"

```
RegisterOpt<ShootMagicJuJu>
    X("shootmagicjuju", "Shoot Magic JuJu on code");
```

- El optimizador las puede cargar y la muestra disponible

```
$ opt -load libshootmagicjuju.so -help
...
-sccp            - Sparse Conditional Constant Propagation
-shootmagicjuju  - Shoot Magic JuJu on code
-simplifycfg     - Simplify the CFG
...
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un programa inocente

```
define i32 @sum (i32 %n) {
entry: %sum = alloca i32
       store i32 0, i32* %sum
       %i = alloca i32
       store i32 0, i32* %i
       br label %lab1
lab1:  %t1 = load i32* %i
       %t2 = add i32 %t1, 1
       %t3 = load i32* %sum
       %t4 = add i32 %t2, %t3
       store i32 %t2, i32* %i
       store i32 %t4, i32* %sum
       %t5 = icmp eq i32 %t2, %n
       br i1 %t5, label %end, label %lab1
end:   ret i32 %t4
}
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un programa inocente

```
define i32 @sum (i32 %n) {
entry: %sum = alloca i32
       store i32 0, i32* %sum
       %i = alloca i32
       store i32 0, i32* %i
       br label %lab1
lab1:  %t1 = load i32* %i
       %t2 = add i32 %t1, 1
       %t3 = load i32* %sum
       %t4 = add i32 %t2, %t3
       store i32 %t2, i32* %i
       store i32 %t4, i32* %sum
       %t5 = icmp eq i32 %t2, %n
       br i1 %t5, label %end, label %lab1
end:   ret i32 %t4
}
```

$$sum = \sum_{i=0}^{n} i$$

UNIVERSIDAD SIMÓN BOLÍVAR

Dejamos que LLVM haga lo suyo

Después de aplicarle `opt -std-compile-opts`

```
define i32 @sum(i32 %n) nounwind readnone {
entry:
  %tmp3 = add i32 %n, -2
  %tmp1 = add i32 %n, -1
  %tmp4 = zext i32 %tmp3 to i33
  %tmp2 = zext i32 %tmp1 to i33
  %tmp5 = mul i33 %tmp2, %tmp4
  %tmp6 = lshr i33 %tmp5, 1
  %tmp7 = trunc i33 %tmp6 to i32
  %tmp = shl i32 %n, 1
  %tmp8 = add i32 %tmp, %tmp7
  %tmp9 = add i32 %tmp8, -1
  ret i32 %tmp9
}
```

UNIVERSIDAD SIMÓN BOLÍVAR

Dejamos que LLVM haga lo suyo

Después de aplicarle `opt -std-compile-opts`

```
define i32 @sum(i32 %n) nounwind readnone {
entry:
  %tmp3 = add i32 %n, -2
  %tmp1 = add i32 %n, -1
  %tmp4 = zext i32 %tmp3 to i33
  %tmp2 = zext i32 %tmp1 to i33
  %tmp5 = mul i33 %tmp2, %tmp4
  %tmp6 = lshr i33 %tmp5, 1
  %tmp7 = trunc i33 %tmp6 to i32
  %tmp = shl i32 %n, 1
  %tmp8 = add i32 %tmp, %tmp7
  %tmp9 = add i32 %tmp8, -1
  ret i32 %tmp9
}
```

$\sum_{i=0}^{n} i = n(n+1)/2$ – Are you scared?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo puedo aprovechar LLVM en mi lenguaje?

- Llamar directamente a las librerías LLVM desde el *front-end*
  - No es afectado por cambios en LLVM IR ni representación interna.
  - Permite ejecutar las pasadas de mejoramiento de manera directa.
  - Soporte inmediato para JIT.
  - Código **horroroso** y **doloroso**.
- Emitir LLVM IR desde el *front-end*.
  - Necesidad de adaptarlo si cambia el LLVM IR.
  - El *back-end* `.ll` de LLVM es más lento que el de *bytecode*.
  - Código **sencillo** y **obvio**.
- Emitir LLVM *bytecode* desde el *front-end*.
  - Mucha adaptación si cambia el LLVM *bytecode*.
  - El *back-end* `.bc` de LLVM es más rápido que el de IR.
  - Código para repetir la conversión entre IR y *bytecode*.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de código estático

- `llc` compila LLVM IR hacia código de máquina nativo.

```
$ llc foo.bc -o foo.s -march=x86
$ as foo.s -o foo
```

- `llc` compila LLVM IR hacia C portátil.

```
$ llc foo.bc -o foo.c -march=c
$ gcc foo.c -o foo
```

- Los lenguajes destino son modulares y dinámicos.

```
$ llc -load libmips.so foo.bc -march=mips
```

UNIVERSIDAD SIMÓN BOLÍVAR

Ejecución de código dinámico

- `lli` permite ejecutar archivos LLVM Binarios

```
$ lli foo.bc ...
```

- Usa el compilador Just-In-Time si está disponible
  - Mismo generador de código que `llc`.
  - Emite código de máquina a memoria directamente.
  - La librería JIT puede embeberse en otras herramientas.
- El interpretador estándar es simple y lento – ¡pero portátil!

UNIVERSIDAD SIMÓN BOLÍVAR

Productos de LLVM

- **CLANG**
  - Front-end C/C++/Objective-C
  - Reconocedor recursivo descendente construido a mano.
  - Soporte completo para C98/C99 y C++98.
  - Soporte casi completo para C++11
  - FreeBSD lo usa como compilador por defecto.
- CLANG vs. GCC
  - Licencia BSD más liberal que la GPL.
  - Código fuente de CLANG es más moderno y limpio.
  - Mejor soporte para CUDA, OpenCL y tecnología similar.
- En Haskell
  - GHC (Glasgow Haskell Compiler) puede generar código vía LLVM.
  - `llvm` – DSL para generar código LLVM desde Haskell.
  - `synthetizer-llvm` – síntesis de audio en tiempo real usando LLVM.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- **The Architecture of Open-Source Applications**
  Elegance, Evolution and a Few Fearless Hacks
  Capítulo 11 – LLVM
- LLVM.org
- LLVM 2.0

UNIVERSIDAD SIMÓN BOLÍVAR