Una Herramienta para chequeo de tipos en Assembly X86

Tesis presentada como requisito parcial
para optar al grado de
Licenciado en Ciencias de la Computación

Facultad de Matemática, Astronomía, Física y
Computación
Universidad Nacional de Córdoba

Autor: Alvaro Frias Garay

Directores: Nicolas Wolovick, Daniel Gutson

Octubre 2023

Índice general

Capítulo 1

Introducción

1.1. Chequeadores de tipos

Un chequeador de tipos es un componente integral en lenguajes de programación asegurando la corrección de código y su fiabilidad. Cuando el software crece en complejidad, es importante también detectar y prevenir errores relacionados a tipos en etapas tempranas de desarrollo.

El objetivo primario de un verificador de tipos es garantizar la seguridad de tipos, lo cual implica asegurar que las operaciones se realicen solo en valores de tipos compatibles y que los errores relacionados con los tipos se detecten antes de la ejecución del programa. Mediante la realización de una verificación de tipos estática, el verificador de tipos puede identificar posibles problemas, como asignaciones incompatibles, desajustes de tipos en llamadas a funciones u operaciones en tipos no admitidos. Esto ayuda a prevenir una amplia gama de errores que pueden provocar excepciones en tiempo de ejecución, corrupción de datos o comportamiento inesperado.

1.2. Planteamiento del Problema

El principal problema a afrontar en este trabajo es lograr tipar un lenguaje de assembly de uso real, en este caso X86. Si bien ya existen formalizaciones e implementaciones de lenguajes tipados [4] [10], el objetivo no es formalizar un assembler de acuerdo a una arquitectura, sino tomar una arquitectura ya existente y dotarla de anotaciones de tipo, en adelante type hints, a sus registros con ciertos tipos definidos y crear reglas para cada familia de instrucciones de la arquitectura.

1.3. Objetivos

El principal objetivo es crear una herramienta que pueda ser usada en una toolchain como herramienta para que el desarrollador de código assembly, i.e. código assembly escrito a mano, no generado previamente por un compilador ni ningún otro programa, sea capaz de usar *type hints* para asegurar la correctitud de los datos en los registros al momento de programar.

1.4. Antecedentes

En cuanto a lenguajes de ensamblador se refiere, ejemplos de estos lenguajes con tipado puede referirse a:

- SIFTAL: Una formalización de lenguaje assembler enfocado en seguridad, más específicamente que se cumplan propiedades de no-interferencia [4].
- TALx86: Una implementación de un subconjunto de X86 de 32 bits. Esta formalización tiene también una implementación de un lenguaje c-like llamado Popcorn, el cuál genera assembly de TalX86 [10].

Remarcando otros lenguajes que no cuentan con un sistema de tipos incorporado de manera nativa sino que se agregan anotaciones son de notar los siguientes casos:

- mypy: una herramienta de tipado estático para Python, la cuál sirve agregando *type hints* al código de python y llamando a la herramienta antes de usar el intérprete. Los tipos son ignorados por el intérprete de Python en tiempo de ejecución [1].

  Un ejemplo de código Python sin type hints:

```
    def add_numbers(a, b):
        return a + b

    result = add_numbers(5, 3)
    print(result)

    # Esto va a funcionar pero no tiene type safety:
    result2 = add_numbers("Hello", "World")
    print(result2) # "Hello World"
```

  con type hints:

```
    def add_numbers(a: int, b: int) -> int:
        return a + b

    result = add_numbers(5, 3)
    print(result)

    # Esto va a producir un error porque estamos tratando de
        sumar dos strings
    result2 = add_numbers("Hello", "World")
    print(result2)
```

- flow: Similar a mypy, flow es una herramienta de tipado estático que sirve para realizar anotaciones de tipos para Javascript [2].

  Un ejemplo de código Javascript sin tipar:

```
function addNumbers(a, b) {
    return a + b;
}

const result = addNumbers(5, 3);
console.log(result);

// Esto va a funcionar pero no tiene type safety:

const result2 = addNumbers("Hello", "World");
console.log(result2);
```

  con tipos:

```
// @flow

function addNumbers(a: number, b: number): number {
    return a + b;
}

const result = addNumbers(5, 3);
console.log(result);

// Esto va a producir un error de Flow
// porque tratamos de sumar dos strings
const result2 = addNumbers("Hello", "World");
console.log(result2);
```

- mopsa: Chequeador de tipos para Python que hace uso de técnicas de abstract interpretation para detectar errores. [9]

1.5. Sistemas de tipos

Antes de continuar haremos una breve introducción a las diferentes características presentes en los sistemas de tipos, y trataremos de definir cómo será el sistema de tipos que implementaremos en este trabajo.

Tipado estático y dinámico

Estas dos propiedades se refieren al momento al cual se definen los tipos de las variables del programa.

En el caso del tipado estático, el tipo de la variable es especificado en tiempo de compilación. Esto significa que una vez declarada la variable con cierto tipo, no se puede cambiar su tipo luego en el programa. Esto ayuda a detectar errores tempranos durante la fase de compilación. Ejemplos de lenguajes con tipado estático incluyen C, C++, Java.

En el tipado dinámico, el tipo de una variable es determinado en tiempo de ejecución. Esto provee flexibilidad dado que una variable puede cambiar su tipo durante la ejecución del programa. Sin embargo, puede conducir a errores en tiempo de ejecución si hay tipos que no son compatibles. Ejemplos de tipado dinámico incluyen Python, Ruby y Javascript.

Tipado Fuerte y Débil

En lenguajes con tipado fuerte, una vez una variable ha sido declarada con un tipo, cualquier operación sobre ella que no es aplicable a ese tipo producirá un error. Esto asegura seguridad de tipos pero puede ser restrictiva. Ejemplos son Java y Python.

Mientras, los lenguajes débilmente tipados son más indulgentes. Permiten operaciones sobre variables incluso si la operación no tiene el mismo tipo que la variable, a menudo realizando conversión implícita de tipos. Esto puede llevar a resultados inesperados. Javascript es un ejemplo de esto.

Tipado Explícito e ímplicito

El tipado explícito requiere que el programador declare el tipo de una variable cuando se está definiendo. Esto proporciona claridad sobre el tipo de dato que una variable puede contener. Java, donde se declara una variable como ‘int x = 10;‘, es un ejemplo.

En el tipado implícito el entorno de ejecución determina el tipo de la variable según el valor asignado a ella. Esto puede hacer que el código sea más corto y legible. En Python, simplemente se puede escribir ‘x = 10‘, y el intérprete sabe que ‘x‘ es un entero.

Tipado Nominal e implícito

El tipado nominal significa que dos variables son del mismo tipo sólo si están declaradas con el mismo nombre de tipo. Un ejemplo de lenguaje con este tipado es Java.

En el tipado estructural la compatibilidad y equivalencia de tipos se determinan por su estructura, no por sus nombres. TypeScript es un buen ejemplo, donde dos tipos con nombres diferentes pueden considerarse iguales si tienen propiedades idénticas

Tipado Duck

Este concepto se basa en el dicho: ”Si parece un pato, nada como un pato y grazna como un pato, entonces probablemente es un pato”. En programación, significa que el tipo de un objeto se determina por su comportamiento (métodos y propiedades) en lugar de su clase o herencia. Para esto es necesario que el sistema de ejecución lleve registro de los tipos de los elementos. Python es un lenguaje que emplea el tipado duck.

Tipado Genérico

Los genéricos permiten parámetros de tipo, lo que hace posible diseñar clases, interfaces y métodos que funcionen de manera agnóstica al tipo. Al usarlos, se pueden especificar los tipos exactos. Los genéricos de Java, donde se puede tener una List$< T >$, es un ejemplo.

Inferencia de Tipo

Esta es la capacidad de un lenguaje para deducir el tipo de un valor en contextos donde no se proporciona un tipo explícito. Combina la seguridad del tipado estático con la facilidad del tipado dinámico. Haskell es conocido por su sistema de inferencia de tipo.

Si bien existen varias otras clasificaciones de tipos nos centraremos en estas para poder definir el sistema de tipos de nuestra herramienta.

Soundness

Soundness se refiere a una característica de los sistemas de tipos donde un sistema de tipo es sound si garantiza que las operaciones que son incorrectas para un tipo no serán realizadas.

De otra manera, un sistema de tipos será sound si y solo si programas tipados correctamente no producen errores de tipos durante la ejecución.

El chequeo de esta propiedad la veremos más adelante.

Definición del sistema

Para este trabajo, definimos el sistema de tipos de la siguiente manera: primero, los tipos serán definidos sobre los registros de la arquitectura X86 y el stack de memoria en una función, teniendo como posibles tipos Number o Pointer, o lo que es lo mismo, tener un dato o puntero a un dato.

El sistema deberá ser de tipado estático, donde los chequeos de tipos ocurrirán antes de que el código ensamblador sea transformado a código máquina.

Para asegurar seguridad en los tipos, el sistema será fuertemente tipado. Es decir que si, por ejemplo, un registro es tipado como un número, entonces usarlo como puntero o en una instrucción que espere un número provocará un error de tipos.

Dada la naturaleza del lenguaje assembly, es mejor utilizar el tipado explícito para claridad. type hints serán usadas como directivas antes de las instrucciones o junto a declaraciones de registros.

Comparando entre nominal y estructural, nuestro sistema de tipos sería principalmente nominal. Cada type hint correspondería a un tipo con un nombre específico (por ejemplo, "number", "pointer").

El tipado Duck no es aplicable dado que no hay un sistema de runtime capaz de llevar registro de los tipos de datos. En este caso la ejecución la realiza directamente el procesador.

Tampoco aplica el tipado genérico, puede ser tratado como un trabajo futuro.

Finalmente, la inferencia de tipos puede ser usada de manera limitada basada en el uso de los registros y la memoria. En específico trabajaremos con el caso de la familia de instrucciones mov y lea. Donde una inferencia de tipos es casi directa dado que lo transferirá al registro de destino.

Con todas las definiciones ya listadas, podemos continuar con el foco de esta tesis.

1.6. Ejemplo motivador

A continuación se mostrará un ejemplo de un programa en Assembler X86, *ejemplo.asm*, con el que motivar el tipado posterior dada la complejidad de los datos que se manejan en los registros del programa logrando reducir así la cantidad de errores posibles al momento de escribir un programa en assembler a mano.

```
1  strcpy:
2          movq    %rdi, %rax
3          movq    %rsi, %rdx
4          movq    %rax, -8(%rbp)
5          jmp     .L2
6  .L3:
7          addq    $1, %rdx
8          addq    $1, %rax
9  .L2:
10         movzbl  (%rdx), %ecx
11         movzbl  (%rax), %ecx
12         jne     .L3
13         movq    -8(%rbp), %rax
14         ret
```

El programa listado anteriormente se trata de una implementación de strcpy, es decir de copiar una string de un registro dado, en este caso rdi, a otro de destino, rdi. Para esto la función espera que los registros del stack sean dos punteros.

Capítulo 2

La herramienta

Desarrollaremos un chequeador de tipos para assembler X86 que permita agregar type hints en los registros de las instrucciones. Para desarrollar la herramienta utilizaremos una infraestrutura de compilador, en adelante toolchain, ampliamente usado industrialmente: LLVM.

2.1. LLVM

LLVM es una toolchain utilizada ampliamente en la industria que proporciona una colección de tecnologías de compilación modulares y reutilizables. Fue desarrollado inicialmente como un proyecto de investigación en la Universidad de Illinois y desde entonces se ha convertido en un proyecto de código abierto ampliamente adoptado y activamente mantenido.

En su núcleo, LLVM ofrece un marco versátil para construir compiladores, lo que permite a los desarrolladores transformar de manera eficiente lenguajes de programación en código ejecutable. Utiliza una representación intermedia (IR, por sus siglas en inglés) llamada LLVM IR, que sirve como una representación portable e independiente de la plataforma de programas. Esta IR está diseñada para ser fácilmente optimizada y transformada para generar código eficiente para una amplia gama de arquitecturas de destino, incluyendo CPU, GPU e incluso hardware especializado.

Una de las características clave de LLVM es su énfasis en la optimización. Proporciona un conjunto completo de pasadas de optimización que pueden analizar y transformar el IR para mejorar el rendimiento del código. Estas optimizaciones incluyen técnicas comunes como la reducción de constantes, el desenrollado de bucles y la expansión de funciones, así como técnicas más avanzadas como la vectorización automática y la optimización guiada por desempeño de ejecución. El diseño modular de LLVM permite a los desarrolladores personalizar y ampliar fácilmente estas pasadas de optimización según sus necesidades específicas.

Además de la optimización, LLVM también admite una variedad de otras ta-

reas relacionadas con la compilación. Incluye un potente compilador de tiempo de ejecución (JIT) que permite la compilación y ejecución dinámica de código en tiempo de ejecución. Esto hace que LLVM sea adecuado para implementar lenguajes de programación dinámicos o optimizar secciones críticas de rendimiento en aplicaciones. LLVM también proporciona herramientas para análisis estático, depuración y generación de código, lo que lo convierte en una solución integral para el desarrollo de compiladores.

Con respecto a lo que queremos lograr en este trabajo, LLVM contiene un módulo dedicado especificamente a tareas relacionadas a manejo de ensamblador: llvm-mc.

2.1.1. llvm-mc

llvm-mc es una herramienta del toolchain de LLVM que sirve para recibe código assembly de una arquitectura dada y genera archivos de objeto o ejecutables para una determinada arquitectura. Esta herramienta está desarrollada, así como el resto de LLVM, en el lenguaje C++.

La herramienta desarrollada en esta tesis hace uso intensivo del módulo MC de LLVM para, por ejemplo, si tomamos el programa de la sección anterior y queremos ensamblarlo mostrando cómo se encodea a la arquictura deseada, X86 en este caso, se utilizaria un comando como el siguiente:

```
1    llvm-mc -assemble -show-encoding -arch=x86  ejemplo.s
```

Como trabajo de esta tesis, expandiremos MC para que sea posiblea chequear tipos.

Comenzaremos por actualizar el módulo de MC para poder recuperar el assembler parseado.

2.1.2. MC

El módulo de MC sirve para operaciones de ensamblado, desensamblado, formateo del file object y otras áreas relacionadas a las instrucciones de CPU.

Las principales clases de MC son:

- MCAsmParser: la clase responsable del parseo del código fuente. Se encarga de la tokenización, parseo y validación de las instrucciones de assembler.
- MCStreamer: Esta clase es usada para emitir código de máquina y datos en un *stream* de salida.
- MCCodeEmitter: Es el responsable de traducir la representación intermedia al código de máquina. Se encarga del encodeo de la instrucción, operandos y generación de bytes.
- MCParsedAsmOperand: Clase principal de un operando de una instrucción. Un vector de McParsedAsmOperand constituye una instrucción.

En esta sección nos centraremos en el parseo de las instrucciones para poder luego trabajar con las instrucciones parseadas.

El módulo hace uso de una clase `AsmParser`, que sirve como clase principal para el parseo de instrucciones de las instrucciones de todas las arquitecturas, cuando es instanciado compone otra clase `MCTargetAsmParser` la cual hace el parseo propiamente de la instrucción de la arquitectura dada.

El parser está implementado haciendo uso de parseo recursivo desecendente para el parseo de los elementos: El entry point y método principal es `run` el cual inicia el lexer. Este es responsable de tokenizar el código fuente assembly, reconocer mnemónico, directivas, registros y otros tokens relevantes, para así pasarlos para su procesado.

Luego comienza el ciclo hasta obtener el símbolo de EOF. En el loop parsea lo que define como `statements`, los cuales pueden definirse mediante la siguiente gramática:

```
1 ParseStatement:
2   ::= EndOfStatement
3   ::= Label* Directive ...Operands... EndOfStatement
4   ::= Label* Identifier OperandList* EndOfStatement
```

El parser ignora espacios en blanco y comienza primero a buscar directivas mediante la primera definición de la gramática, parseando el identificador de la directiva, que por norma son de la forma `.|string|` y obtiene un valor el cual busca de una lista predefinida de directivas comunes a todas las arquitecturas: if, elseif, else directives; Set Directive; Align directive; Error directive, etc. Luego del parseo de directivas comienza el parseo de las instrucciones propiamente dichas.

Para el parseo de las instrucciones la clase llama al método

`parseAndMatchAndEmitTargetInstruction`, el cual principalmente lo que hace es llamar al Target Parser correspondiente mediante `getTargetParser` y a su vez este usa un método privado `ParseInstruction`, si el parseo de la instrucción no arroja errores.

Durante el parseo de la instrucción propiamente primero se realizan chequeos de alineamiento de la arquitectura, seteo de flags, parseo de directivas propias de la arquitectura. El último proceso de esta etapa es matchear la instrucción con el opcode que le corresponde mediante el `MCCodeEmitter` y el manejo de la salida del opcode con el `MCStreamer`.

2.1.3. Cambios aplicados a MCAsmParser de llvm

Dado que la principal información que nos interesa de parte del AsmParser es la lista de instrucciones en assembler parseadas junto con las correspondientes labels, sin hacer uso del Matcher mencionado anteriormente para generar opcodes de los programas y tampoco enviar los datos generadors a alguna salida vía MCStreamer actualizaremos la clase MCAsmparser para que sirva a estos propósitos.

Comenzamos la tarea creando una nueva clase `TypecheckingAsmParser` que hereda directamente de la clase `AsmParser` dado que hará uso de gran parte del cuerpo del método `Run`.

Primero definimos un nuevo tipo de directiva general en `MCAsmParser` la cual servirá para definir los tipos posibles de los registros al momento de iniciar el programa. La sintaxis es la siguiente:

```
1    SetTypeDirective ::= .setttype Register AllowedType
2    Register ::= RAX | RBX | RCX | RDX | RSI | RDI | RBP | RSP | R8
         -R15 | RIP
3    AllowedType ::= pointer | number
```

Para esto actualizamos en el cuerpo de `Run` donde se parsean las directivas añadiendo el caso dado de .settype y creadndo un nuevo método `parseDirectiveSetType`.

El siguiente paso es aplicar un template method[7] sobre `MCAsmParser`. Definiremos dos template method principalmente para `MCAsmParser`: uno para las label de `.settype` y otro para el parseo de las instrucciones de X86.

El primero se trata de `handleSetTypeDirective`, el cual se implementará en `TypecheckingAsmParser` y servirá cuando el parser detecte la directiva y será guardada en un vector propio para luego ser utilizado.

En el segundo se crea un método virtual `handleParseInstruction` reemplazando el actual `parseAndMatchAndEmitTargetInstruction` y haciendo que esté adentro del mismo `handleParseInstruction` y creando un método `parseAndStoreParsedInstruction` para `TypecheckingAsmParser` que servirá para parsear la instrucción del target parser dado `getTargetParser().ParseInstruction()`. Dado que la información de la instrucción queda almacenada como un vector de `MCParsedAsmOperand`, actualizamos dicha clase con un método `storeValue` con el cuál podremos obtener la información de cada uno de los operandos y almacenarlos en un vector que representa la instrucción en su totalidad.

De esta forma hemos podido actualizar el parser de llvm para poder obtener una representación del programa en assembler con las directivas de seteo de tipos y el programa parseado con sus instrucciones.

El ejemplo en 1.6 con las nuevas directivas de seteo de tipos quedaría de la siguiente forma:

```
1        .settype rdi, "pointer"
2        .settype rsi, "pointer"
3    strcpy:
4            movq    %rdi, %rax
5            movq    %rsi, %rdx
6            movq    %rax, -8(%rbp)
7            jmp     .L2
8    .L3:
9            addq    $1, %rdx
10           addq    $1, %rax
11   .L2:
12           movzbl  (%rdx), %ecx
```

```
13          movzbl  (%rax), %ecx
14          jne     .L3
15          movq    -8(%rbp), %rax
16          ret
```

Figura 2.1: Diagrama de clase original de llvm con los principales componentes en el proceso de parseo

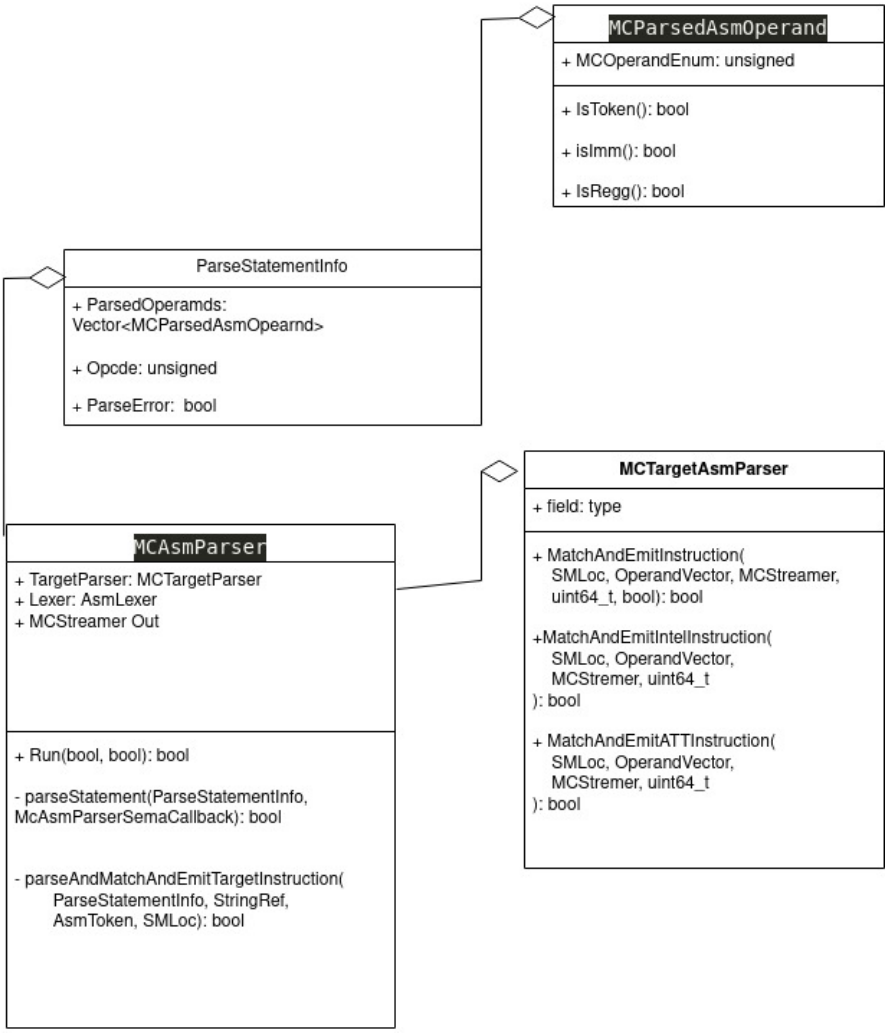
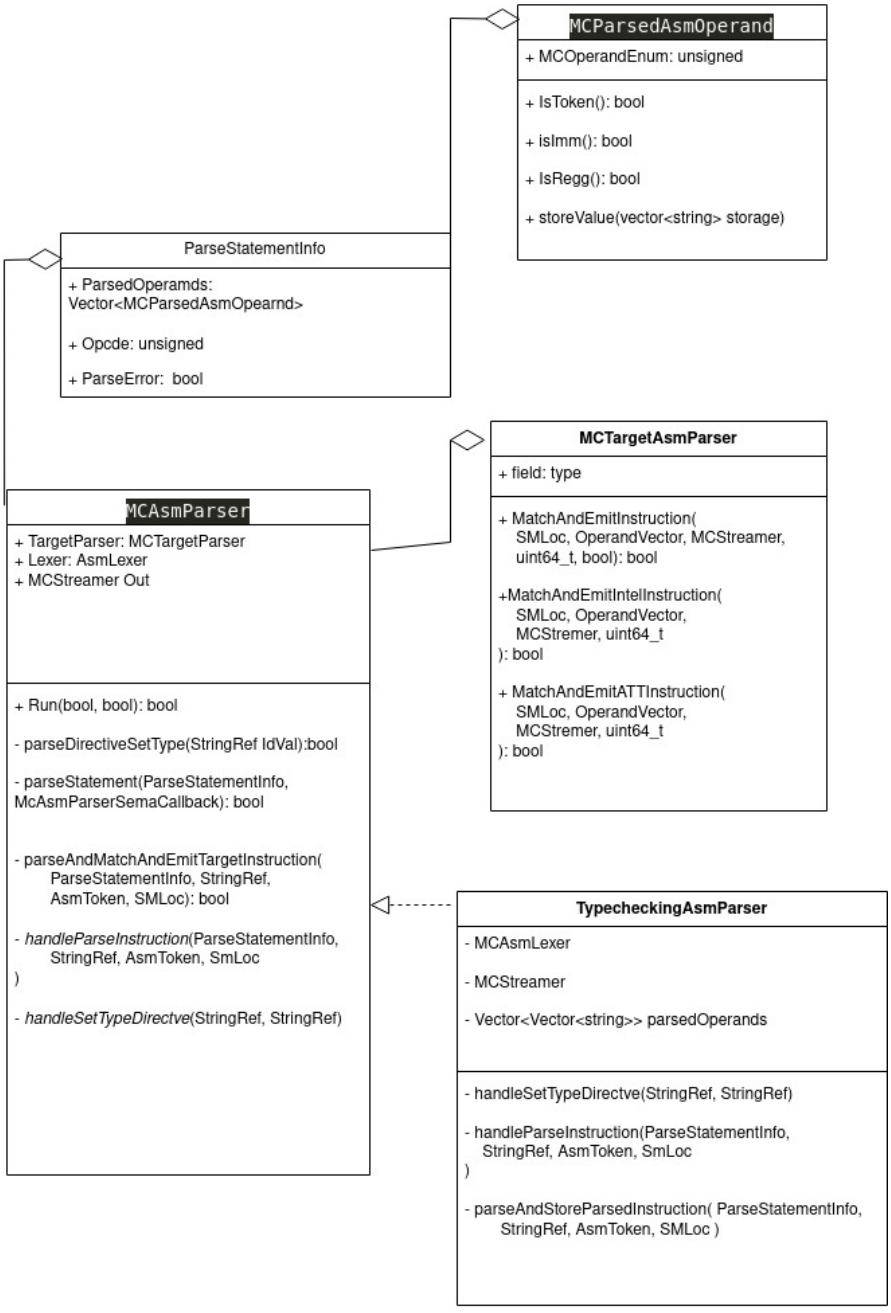


Figura 2.2: Diagrama de clase actualizado de llvm con la clase TypecheckingAsmParser



Capítulo 3

QInstrucciones

Una vez que obtenemos el programa assembler parseado nos topamos con una serie de inconvenientes que dificultan el chequeo de tipos del programa; primero, que las instrucciones provienen de arquitecturas específicas, muchas veces con instrucciones propias de la arquitectura que no son comunes al resto de las arquitecturas existentes, también presentándose en las instrucciones información sobre la arquitectura que tampoco interesa al chequeo de tipos. Otro problema a resolver es la semántica de las instrucciones. Dado que en este caso particular estamos tratando con X86, esta arquitectura tipo CISC presenta instrucciones en cuya semántica se realizan varias operaciones distintas [3], las cuales podrían ser divididas en varias instrucciones mas primitivas. Sin contar con que las instrucciones cuentan con información tampoco nos interesan y de las que podemos abstraernos, e.g. dada una instrucción de X86 como

```
1         movl    -4(%rbp), %eax
```

En esta instrucción se especifica que es un MOV de un registro y, dadp que estamos centrándonos en la notación AT&T, estas tienen un sufijo para denotar el tamaño de los operandos. En este caso, "lreferencia que se trata de una operación de MOV de 32-bits. La notación hace uso de otros sufijos: b para denotar un byte, por ejemplo MOVB; w, para denotar 16-bits; q, para denotar operaciones 64 bits. Dicha información es útil para un compilador pero en este caso no nos interesa, dado que solo nos interesa la semántica de la instrucción.

Para resolver todos estos problemas creamos una arquitectura virtual Q, la cuál cuenta con *QInstrucciones* y registros propios, que recibirá el programa parseado de llvm y lo transformará a un *QPrograma* apto para continuar con el chequeo de tipos.

Dado que es una abstracción de una arquitectura real como X86 u otras, solo existirá como representación intermedia en memoria pero para clarificar mejor el funcionamiento en este trabajo definiremos una gramática para cada uno de las instrucciones junto con las transformaciones realizadas y un ejemplo del programa presentado en Ejemplo motivador.

3.0.1. QRegisters

Para la representación de los registros de la arquitectura definiremos una notación estándar: Todos los registros serán de la forma r_n. donde $n \in \mathbb{N}_0$.

Definiremos un mapeo de los registros de X86 a QArquitectura el cuál a medida que se vayan conviertiendo cada una de las instrucciones del programa se haga un chequeo si ya existen en el mapeo los registros usados o bien si se agrega una nueva entrada al mapeo definiendo el nuevo registro como el r_{k+1}, donde k era el largo del mapeo hasta entonces. Este mapeo también aplicará Para el caso de registros definidos en stack por cada función, por ejemplo rbp-16; rbp-8; rbp-32, etc.

Para simplificar el mapeo dejaremos fijos una serie de registros de X86: rsp, el stack pointer register; rax, el registro el cual es usado implícitamente en varias instrucciones que son consideradas aquí; los inmediatos. Para el caso de los inmeadiatos modificamos el parseo en llvm para que transforme cualquier literal a una abstracción *Imm*.

Una vez consideradas estas definiciones el mapeo inicial al momento de comenzar la transformación del programa a Qprograma sería el siguiente

Registro Original	**QRegistro**
rsp	r0
rax	r1
Imm	r2

Una simplificación realizada sobre los registros es referida al manejo del stack de variables locales. X86 utiliza el registro rbp para mantener el puntero base, el cual mantiene un stack por cada función que se llama. En nuestro caso para cada definición de variable local en el stack consideraremos el puntero y su desplazamiento como un registro en el QPrograma. Es decir, si se tiene un programa donde se define una variable local en una función como $rbp - n$, será transformada a r_i en el QPrograma.

3.0.2. QInstrucciones

A continuación definiremos un set básico de instrucciones para la arquitectura virtual que serán la transformación final de las instrucciones X86 que vendrán del parser de llvm, las cuales son una biyección casi directa a una instrucción presente en X86. Luego presentaremos las transformaciones de instrucciones más complejos a varias QInstrucciones.

QSetType

Gramática:

```
1    QSetType :== Qsettype QRegister RegType
2    AllowedType ::= pointer | number
```

Instrucción para representar el seteo del tipo de una instrucción. Es una biyección con la directiva de seteo de tipos definido en Cambios aplicados a MCAsmParser de llvm.

Ejemplo:
Dada una directiva de X86 como

```
.settype rax "number"
```

su traducción a QInstrucción es

```
QSetType r1 "number"
```

QAdd

Gramática:

```
QAdd :== QAdd QRegister QRegister
```

Instrucción biyectiva a una instrucción de Add de X86.
Ejemplo: Dada una instrucción de X86 como la siguiente

```
add %rax, %rbx
```

Su traducción a QInstruction es

```
QAdd r1, r2
```

QSub

Gramática:

```
    QSub :== QSub QRegister QRegister
```

Instrucción biyectiva a una instrucción de Sub de X86.
Ejemplo:

```
sub %rax, %rbx
```

Su transformación a QInstrucción es

```
Qsub, r1, r2
```

QMul

Gramática:

```
QMul := QMul QRegister r1
```

Biyección a instrucción mul de X86.
Dada la instrucción en X86

```
mul rbx
```

su transformación a QInstrucción es

```
QMul r2, r1
```

donde r1 es el registro correspondiente a rax en X86.

QLogicalOp

Esta instrucción rompe con la biyección con una instrucción de X86 dado que es una representación de varias instrucciones que tienen la misma semántica. En este caso se trata una generailización de las instrucciones lógicas de X86 correspondientes a xor, or, and, not, shr, shl, sar, sal.

Gramática:

```
QLogicalOp := QLogOp QReg QReg
```

Ejemplo: Dada la instrucción de X86

```
and rax tbx
```

su transformación a QInstrucción es tal

```
QLogOp r1, r2
```

QMov

De nuevo una biyección directa, en este caso con mov. La particularidad de esta instrucción es que también guarda información si hubiera operaciones de indirección sobre los registros.

Gramática:

```
QMov := Qmov QReg MemOp QReg MemOp
MemOp := bool
```

Dado una instrucción de X86 como

```
mov %(rax) %rbx
```

Se transforma a su correspondiente QInstrucción como

```
QMov r1 true r2 false
```

QJmp

Instrucción que abstrae la familia de saltos presente en X86: jne, je, jg, jge, jl, jle, jmp, etc. Esto dado que obviaremos lógica de flujo de instrucciones-

Gramática:

```
QJmp := Qjmp Label
Label := string
```

Por ejemplo, si se tiene la instrucción

```
jmp labelx:
```

Se transformará a

```
QJmp labelx
```

QLabel

Enlazado con la instrucción anterior QLabel mantiene información de los labels presente en el programa.

```
QLabel := Qlabel label
label := string
```

La biyección con una label de un programa X86 es obvia.

3.0.3. Transformación de instrucciones complejas

Dado que estamos trabajando con la arquitectura CISC de X86 hay instrucciones las cuales realizan varias operaciones en la misma instrucción. Para simplificar el uso y el proceso de tipar las instrucciones es que también transformaremos estas instrucciones a varias QInstrucciones para simplificar el programa a tipar y que la interfaz sea común al resto de las arquitecturas.

LEA

Para el caso de esta instrucción la expansión a QInstrucciones dependerá de si en el registro src se hace una operación de derefencia y se agrega un operador de index que sirve como multiplicador del registro base. Para los fines de este trabajo solo consideraremos el caso en que se agreguen índices de Inmediatos, no registros.

Dada la instrucción en X86 con el caso en que se agreguen operadores de multiplicación

```
lea 1(%rax), %rbx
```

Definiremos primero la transformación a instrucciones más simples, las cuales serán:

```
mul %rax
add Imm, %rax
mov (%rax), %rbx
```

La transformación a QInstrucciones sería la siguiente, suponiendo que %rbx sea mapeado a r3:

```
Qmul r1
Qadd r3, r1
Qmov r1. true, r3, false
```

Para el segundo caso

```
lea (%rax), %rbx
```

La transformación a instrucciones más simples queda definida como

```
add Imm, %rax
mov (%rax), %rbx
```

Y su transformación a QInstrucciones, suponiendo nuevamente que a %rbx se le mapea r3:

```
1     Qadd r2, r1
2     Qmov r1, true, r3, false
```

En X86 hay conjuntos de instrucciones que quedarán fuera del foco de esta tesis: instrucciones SIMD, e instrucciones de Hardware.

Las instrucciones SIMD (Single Instruction, Multiple Data), refiere a una familia de instrucciones que permiten que una sola operación sea realizada en múltiples datos simultaneamente, siendo particularmente útil en datasets grandes. Ejemplos de estas instrucciones son MMX, SSE y AVX.

Otra familia de instrucciones que tampoco será incluida en este trabajo es la centrada en el hardware, en este caso nos referimos a las instrucciones INP y OUTP, las cuales sirven para interactuar con dispositivos de hardware.

3.0.4. Transformación de ejemplo a QPrograma

A continuación se mostrará un ejemplo de la transformación, dado el programa en 1.6. Dadas las reglas definidas en las secciones anteriores la transformación a QPrograma queda definida como

```
1     QSetType r3 pointer
2     QSetTtyoe r4 pointer
3     QLabel strcpy
4     QMov r3 false r1 false
5     QMov r4 false r5 false
6     QMov r1 false r6 false
7
8     QJmp .L2
9     QLabel .L3
10
11    QAdd r2 r5
12    QAdd r2 r1
13
14    QLabel .L2
15
16    QMov r5 true r7 false
17    QMov r1 true r7 false
18
19    QJmp .L3
20
21    QMov r6 false r1 false
22    QNop
```

con un mapeo de los registros de X86 a QRegistros definidos, según las definiciones dada en 3.0.1 como

```
1     Imm -> r2
2     ecx -> r7
3     rax -> r1
4     rbp-8 -> r6
5     rdi -> r3
6     rdx -> r5
```

```
7       rsi -> r4
8       rsp -> r0
```

Capítulo 4

Grafo de programa y Abstract interpretation

La última etapa de las transformación consta de llevar el QPrograma a un grafo de programa apto para ser tratado con abstract interpretation. Ya hemos visto que con la transformación anterior que la mayor parte de la semántica del programa se ha perdido (simplificación de las instrucciones de jmp a una sola genérrica, pérdida de instrucciones lógicas generalizadas a una sola general, abstracción de literales a una representación fija, etc.), todo esto con el objetivo de centrarnos únicamente en la semántica de los tipoes de los registros y su uso y las instrucciones.

Para poder realizar este análisis haremos uso de un framework de análsis estático que hace uso de Abstract Interpreatation: SPARTA.

4.0.1. SPARTA

Para realizar el análisis estático del grafo de programa haremos uso de un framework llamado SPARTA. El cual es una libreria de componentes para construir analizadores estáticos basado en la teoría de abstract interpretation. El propósito de SPARTA es proveer la ingeniería de Abstract interpretation proveyendo componentes de software con una API simple para poder ensamblar un analizador estático [11]. SPARTA es usado en el optimizador de bytecode de Android ReDex[1].

SPARTA requiere de 4 elementos para poder realizar el análisis estático que se require: el dominio abstracto y el lattice del mismo; la definición del grafo de programa mediante una interfaz y las reglas de análisis de cada nodo del grafo. Pero antes de continuar con la implementación propiamente dicha introduciremos conceptos útiles de Abstract Interpretation.

[1] https://github.com/facebook/redex

4.0.2. Abstract Interpretation

Abstract interpretation es una método para aproximar el comportamiento de sistemas de cómputo. Provee una manera sistemática de derivar sobre aproximaciones o bajo aproximaciones correctas de posibles comportamientos de un sistema. La aproximación está basada en la idea de interpretar las operaciones del sistema en un dominio abstracto más que uno concreto, donde dominio abstracto refiere a una estructura matemática, en este caso un *lattice* que captura una versión simplificada del comportamiento del sistema. [6].

Dominio abstracto

Los dominios abstractos proveen una manera de representar un conjunto infinito de estados concretos con un conjunto finito de estados abstractos. Algunos ejemplos notables:

- Intervalos

  Si se tiene un conjunto de variables que pueden tomar ciertos valores numéricos, un intervalo sirve como dominio abstracto de los valores concretos que puede tomar la variable. por ejemplo si tenemos una variable x, la cual puede tomar cualquier valor entre 1 y 10 entonces su dominio abstracto es [1, 10] [5]

- Signos

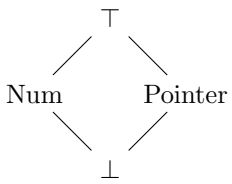
  Si se quiere abstraer variables a signos (positivo, negativo o desconocido), el dominio abstracto definido para las variables son los signos +, -. Por ejemplo, si se tiene una variable y, la cual siempre es positiva, entonces puede ser representada como +.

Lattice

En Abstract Interpreation, los lattices juegan un rol fundacional en la estructura de las relaciones entre los estados abstractos. Un lattice provee una base matemática que captura el orden de los estados abstractos y permite la combinación de estos estados de manera sistemática.

Muchas propiedas de interés pueden ser caraterizadas como puntos fijos sobre lattices. Abstract interpretation a menudo involucra computar estos puntos fijos para determinar propiedades del programa [6]

Con el caso del chequeador de tipos definiremos un dominio abstracto simple pero efectivo a fines del alcance de esta tesis. Definiremos para los registros de las instrucciones definidos en la QArquitectura los tipos *pointer* y *number* para representar los dominios de los tipos posibles de los registros en un momento dado sumado a $\bot$ y $\top$ para definir la incertidumbre de información sobre los tipos en un estado dado del programa. Así es como queda definido el lattice inicial para el chequeador de tipos:



4.0.3. Motor de Punto Fijo

Un punto fijo, en el contexto de matemáticas, refiere al valor que se mantiene cuando una función específica se le aplica. Es decir, para una función f, un valor x es un punto fijo si

$f(x) = x$

En Abstract interpretation, los puntos fijos juegan un rol fundamental a la hora de determinar la conducta de los programas. Especialmente cuando hay bucles o funciones recursivas.

Dado un dominio abstracto, como el que definimos anteriormente, para representar los estados abstractos de un programa y los trasnformadores abstractos, los cuales pueden definirse como el efecto que representa una operación o statement del programa sobre un dominio abstracto, el cual termina generado un nuevo dominio abstracto, Abstract interpretation opera empezando con una pre condición y aplicando transformadores abstractos a cada statement del programa hasta que se alcanza un punto fijo abstracto. Es decir que cualquier posible aplicación de un transformador abstracto no cambia la abstracción

SPARTA cuenta con un Motor de punto fijo el cual provee la funcionalidad necesaria para abstraernos y lo único que requiere es que definamos cómo es el grafo del programa mediante una interfaz y los transformadores abstractos de los statements, en este caso las instrucciones de un QPrograma con los registros conteniendo el dominio abstracto de los tipos.

Representación del grafo de programa

Como dijimos en la sección anterior, SPARTA provee un framework para abstract interpretation pero es necesario modelar el grafo de programa. Definiremos una estructura de bloques, llamados Bloques Básicos, de ahora en adelante BB, los cuales contienen un número de Qinstrucciones, estos servirán como nodos del grafo. Los vértices del grafo quedan definidos por el sucesor y antecesor de la instrucción. Los únicos vértices distintos son los que realizan los BB que contienen instrucciones de Jmp hacía BB que contienen la label correspondiente al salto.

A fines de este trabajo por ahora solo tomaremos una instrucción por BB.

Haremos primero una formalización de una función de QPrograma al grafo de Programa constituido por BB.

Definición: Basic Block (BB)

Un *Basic Block* (BB) es una secuencia de instrucciones con un único punto de entrada y un único punto de salida. Formalmente, un BB se define como:

$$BB = \{\text{Mnemonic}_1, \text{Mnemonic}_2, \ldots, \text{Mnemonic}_n\}$$

Definición: Mnemonic

Un *Mnemonic* es una representación simbólica de una sola instrucción en un BB. Encapsula la información de la instrucción así como sus registros. Formalmente, un Mnemonic se define como:

$$\text{Mnemonic} : \text{Instruction Information}$$

A continuación definimos las funciones de transformación.

f_{reg} : Función que mapea una instrucción i de un $QProgram$ a un bloque básico BB en un $Program$.

$$f_{\text{reg}}(i) = \begin{cases} \text{BB(mov)}(r_1, b_1, r_2, b_2) & \text{si } i \text{ es de tipo } QMov \\ \text{BB(SetType)}(r, \{\text{Pointer}, \text{Number}\}) & \text{si } i \text{ es de tipo } QSetType \\ \text{BB(add)}(r_1, r_2) & \text{si } i \text{ es de tipo } QAdd \\ \text{BB(sub)}(r_1, r_2) & \text{si } i \text{ es de tipo } QSub \\ \text{BB(mul)}(r_1, r_2) & \text{si } i \text{ es de tipo } QMul \\ \text{BB(div)}(r_1, r_2) & \text{si } i \text{ es de tipo } QDiv \\ \text{BB(LopOp)}(r_1, r_2) & \text{si } i \text{ es de tipo } QLogicalOp \\ \text{BB(Nope)}() & \text{si } i \text{ es de tipo } QJmp, QLabel, \text{ o } QNop \end{cases} \tag{4.1}$$

Definición de f_{jmp} :

$\text{f}_{\text{jmp}} : J \times L \to B \times B$

Donde:

- J es el conjunto de instrucciones de salto.
- L es el conjunto de etiquetas.
- B es el conjunto de Bloques Básicos.

La función f_{jmp} se define como:

$$f_{\text{jmp}}(i, l) = \begin{cases} \text{Arista dirigida de } BB(i) \\ \text{a } BB(l) & \text{si } i \text{ es una instrucción de} \\ \text{salto que apunta a la etiqueta } l \end{cases}$$

Donde: - $BB(i)$ es el Bloque Básico correspondiente a la instrucción i.

- $BB(l)$ es el Bloque Básico correspondiente a la etiqueta l.

Las últimas dos funciones auxiliares hacen referencia a los nodos de inicio y salida del programa.

$\text{f}_{\text{primero}} : F \to B$

$\text{f}_{\text{primero}}(i_{\text{primero}}) = BB(i_{\text{primero}})$

Esta función mapea la primera instrucción de un QPrograma a su correspondiente BB en el Programa.

$\text{f}_{\text{ultimo}} : L_{\text{inst}} \to B$

$\text{f}_{\text{ultimo}}(i_{\text{ultimo}}) = BB(i_{\text{ultimo}})$

Esta función mapea la última instrucción de un QPrograma a su correspondiente último BB en el Programa

Definiciones:

1. Q (Conjunto de QPrograms): Este conjunto representa todos los posibles QPrograms. Cada elemento en Q es una secuencia de instrucciones, donde cada instrucción puede ser de tipo `QMov`, `QSetType`, `QAdd`, `QSub`, `QMul`, `QDiv`, `QLogicalOp`, `QJmp`, `QLabel`, o `QNop`.

2. P (Conjunto de Programas): Este conjunto representa todos los posibles Programas en forma de gráficos de flujo de control. Cada elemento en P es una colección de bloques básicos, donde cada bloque básico contiene un mnemónico que representa una instrucción.

Ahora definimos la función de transformación t:

$$\mathrm{t} : Q \to P$$

Ahora, integrando esto en nuestra función de transformación $\mathrm{t} : Q \to P$:

$$\mathrm{t}(q) = p$$

Donde:

$$\begin{aligned} p = \{&f_{\text{reg}}(i) \mid i \in I_q\} \cup \\ &\{f_{\text{jmp}}(i, l) \mid i \in J_q, l \in L_q\} \cup \\ &\{f_{\text{primero}}(i_{\text{primero}}) \mid i_{\text{primero}} \in F_q\} \cup \\ &\{f_{\text{ultimo}}(i_{\text{ultimo}}) \mid i_{\text{ultimo}} \in L_{\text{inst}q}\} \end{aligned}$$

Aquí:

- I_q es el conjunto de instrucciones regulares en QProgram q.
- J_q es el conjunto de instrucciones de salto en QProgram q.
- L_q es el conjunto de etiquetas en QProgram q.
- i_{primero} es la primera instrucción en QProgram q.
- i_{ultimo} es la última instrucción en QProgram q.

Con la función de transformación de QProgramas a grafo de programa tomemos el ejemplo del programa en 3.0.4. Su expresión en forma de grafo conformado por BB sería la siguiente:

Figura 4.1: Representación de grafo de control de Strcpy transformado desde un QPrograma

Cuadro 4.1: heurísticas para Instrucción Add

Dest	**Src**	**Nuevo Tipo Dest**
Int	Int	Int
Pointer	Pointer	Error
Int	Pointer	Pointer
Pointer	Int	Pointer

Cuadro 4.2: heurísticas para Instrucción Sub

Dest	**Src**	**Nuevo Tipo Dest**
Int	Int	Int
Pointer	Pointer	Int
Int	Pointer	Error
Pointer	Int	Pointer

Heurísticas de abstract transformation

Una vez definida la transformación de QPrograma a Programa como instrefaz de grafo el último paso para que SPARTA lo corra es definir las heurísticas de abstract transformation para cada una de las instrucciones definidas en la subsección anterior. A continuación se presentan las heurísticas de los tipos de cada instrucción según la composición de los tipos de las instrucciones.

Soundness del sistema de tipos

Una vez definidos el sistema de tipos queda definir si el mismo es sound o unsound.

Recordemos que un sistema de tipos es sound si y solo si programas tipados correctamente no producen errores de tipos en tiempo de ejecución.

A continuación probaremos que nuestro sistema de tipos es unsound. Para esto se necesita un programa que esté bien tipado, es decir, que pasa el chequeo de tipos, pero que produce un error de tipo durante la ejecución.

Para probar esto antes declararemos una serie de propiedades que el sistema de tipos posee:

- Si un registro no tiene un type hint al inicio del chequeo de tipos, se le es asignado el valor $\top$

Cuadro 4.3: heurísticas para Instrucción Mul

Dest	**Nuevo Tipo Dest**
Int	Int
Pointer	Error

Cuadro 4.4: heurísticas para Instrucción Div

Dest	**Nuevo Tipo Dest**
Int	Int
Pointer	Error

Cuadro 4.5: heurísticas para Instrucción SetType

Dest	**TypeOption**	**Nuevo Tipo Dest**
X	Pointer	Pointer
X	Number	Number

Cuadro 4.6: heurísticas para Instrucción LogOp

Dest	**Src**	**Nuevo Tipo Dest**
Int	Int	Int
Pointer	Pointer	Error
Int	Pointer	Error
Pointer	Int	Error

Cuadro 4.7: heurísticas para Instrucción Mov

Dest	**Src**	**Nuevo Tipo Dest**
X	Int	Int
X	Pointer	Pointer
X	Pointer	Int (si es una operación de derefencia)
X	Int	Error (si es una operación de derefencia)

- A medida que el programa avanza, el tipo de un registro puede ser refinado de $\top$ a Pointer o Number, basado en type hints o uso.
- El tipo de un registro nunca puede ser elevado de nuevo a $\top$ una vez ha sido refinado.

Con estas propiedades declaradas demos un caso donde el sistema de tipos es unsound.

Supongamos un programa, donde se tienen dos registros r1, r2. Al principio, ambos son de tipo $\top$. Basado en alguna instrucción, r1 es refinado a tipo Pointer. Luego en el programa, hay un instrucción que espera que r1 y r2 sean del mismo tipo. En este punto el chequeador de tipos ve a r1 con tipo Pointer y a r2 como tipo $\top$, y permite la instrucción dado que $\top$ es supertipo de todos los tipos. Sin embargo, luego en el programa, r2 es refinado al tipo Number, basado en su uso o mediante la directiva de seteo de tipos. Ahora, tenemos una situación donde r1, un registro de tipo Pointer, y r2, un registro de tipo Number, han sido usados en una instrucción que espera que sean del mismo tipo. Esto es un error de tipos.

Dado el programa definido anteriormente hemos demostrado que es posible para un programa bien tipado (de acuerdo al chequeo de tipos), producir un error de tipos durante su ejecución. Por lo tanto, el sistema de tipos es unsound.

Con todas las heurísticas de los Mnemónicos ya definidas podemos finalmente correr el motor de punto fijo y chequear si el programa está correctamente tipado o no.

Capítulo 5

Resultados

Una vez completada la herramienta con todas sus etapas de transformación del programa y el chequeo de tipos mediante SPARTA presentamos a continuación una serie de programas de prueba sumado al programa de ejemplo 1.6 con distintos tipos sobre registros para comprobar el funcionamiento de la herramienta.

5.1. Programas de prueba y tipos

Pointer Inc

Una función de assembler que toma como parámetro un puntero a un número y lo incrementa en uno.

```
1     f:
2 .settype rdi, "pointer"
3         movq    %rdi, -8(%rbp)
4         movq    -8(%rbp), %rax
5         movl    (%rax), %eax
6         addl    $1, %eax
```

Seteando el tipo de rdi como "number"provoca un error de tipo. A continuación analizamos el chequeo de tipos a detalle instrucción por instrucción para este caso.

Si suponemos el caso donde rdi queda definido como un número en vez de como un puntero, la instrucción

```
1     movq %rdi, -8(%rbp)
```

esta operación es válida y transfiere el tipo de rdi a rbp-8 (número). La siguiente instrucción

```
1     movq -8(%rbp), %rax
```

nuevamente transfiere el tipo, en este caso de rbp-8 a rax. Finalmente en la instrucción

```
    movl (%rax), %eax
```

Es donde surge el error de tipos. Dado que ocurre una deferencia del registro rax pero el tipo es número esta operación es inválida y el chequeo de tipos produce un error.

Suma de punteros

Similar al programa anterior pero en este caso se recibe por parámetro dos punteros.

```
.settype rdi, "pointer"
.settype rsi, "pointer"

f:
        movq    %rdi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        movq    -8(%rbp), %rax
        movl    (%rax), %edx
        movq    -16(%rbp), %rax
        movl    (%rax), %eax
        addl    %edx, %eax
```

Otras combinaciones de tipos en los registros rdi, rsi que incluyan un type hint "number"producen un error de tipos.

Xorshift32

Una variación de Xorshift[8], un generador de números pseudoaleatorios el cual genera números pseudoaleatorios a partir de una semilla utilizando la operación Xor y el shifteo de un bit a la derecha. Este ejemplo recibe como parámetro un puntero a la semilla.

```
    .settype rdi, "pointer"
xorshift32:
        movl    (%rdi), %edx
        movl    %edx, %eax
        sall    $13, %eax
        xorl    %edx, %eax
        movl    %eax, %edx
        shrl    $17, %edx
        xorl    %eax, %edx
        movl    %edx, %eax
        sall    $5, %eax
        xorl    %edx, %eax
        movl    %eax, (%rdi)
        ret
```

Setenado el tipo de rdi como numero produce un error de tipos.

strcpy

El ejemplo de 1.6, se trata de una variación de copiar una string de una variable a otra. Recibe por parámetro dos punteros, donde se copia el contenido del primero al segundo.

```
    .settype rdi, "pointer"
    .settype rsi, "pointer"

strcpy:
        movq    %rdi, %rax
        movq    %rsi, %rdx
        movq    %rax, -8(%rbp)
        jmp     .L2
.L3:
        addq    $1, %rdx
        addq    $1, %rax
.L2:
        movzbl  (%rdx), %ecx
        movzbl  (%rax), %ecx
        jne     .L3
        movq    -8(%rbp), %rax
        ret
```

Las otras combinaciones que incluyan un type hint "number"producen un error de tipos.

Capítulo 6

Conclusiones

6.1. Resumen de Hallazgos

A través de este trabajo, nos hemos encargado de explorar y desarrollar un chequeador de tipos basado en análisis estático, centrado en el contexto de código de ensamblador de X86. La motivación principal era asegurar la corrección de código y su fiabilidad, la cual se vuelve crucial a medida que el código crece en complejidad.

Al aprovechar el poder y la flexibilidad de LLVM, una infraestructura de compilador de vanguardia, nuestro objetivo fue desarrollar una herramienta que se integrara sin problemas en la cadena de herramientas de un desarrollador. Esta herramienta fue diseñada no solo como un ejercicio académico sino como una solución práctica para los desarrolladores, ayudando a aquellos que escriben código ensamblador a mano. El objetivo era ambicioso: no solo formalizar un ensamblador según una arquitectura, sino tomar una arquitectura existente como X86 y dotarla de indicaciones de tipo para sus registros.

Hemos podido desarrollar una herramienta que es capaz de tipar programas escritos es assembler x86 permitiendo tipar assembler con cambios mínimos, solo teniendo que añadir directivas de seteo de tipos al código fuente.

6.2. Contribuciones

A lo largo de este trabajo, hemos realizado contribuciones significativas en el campo de la verificación de tipos y el análisis de lenguajes ensambladores:

Header library

Hemos diseñado y desarrollado una biblioteca de encabezado específicamente para los QPrograms y el motor de punto fijo. Esta biblioteca facilita la integración, interpretación y manipulación de QPrograms y el chequeo de tipos,

proporcionando una base sólida para futuras investigaciones y desarrollos en este ámbito.

Actualización de llvmmc

Cómo se ha tratado en las primeras secciones de este trabajo, hemos actualizado la herramienta llvmmc para que sea capaz de soportar el tipeo de programas de X86. Entre los trabajos futuros se encuentra contribuir esta característica al repositorio principal de llvm.

Extensibilidad a otros Compiladores

A pesar de que nuestra herramienta se basó inicialmente en LLVM, hemos asegurado que sea extensible a otros compiladores. Esta extensibilidad garantiza que nuestra contribución no esté limitada en su aplicabilidad y pueda adaptarse a diferentes plataformas y entornos, gcc por ejemplo.

Al garantizar que nuestra biblioteca y análisis no estén exclusivamente limitados a LLVM, hemos creado una solución versátil que puede adaptarse y evolucionar con las demandas del lenguaje ensamblador.

Extensibilidad a Otras Arquitecturas

Hemos diseñado nuestra herramienta para que sea adaptable a diferentes arquitecturas, como ARM y MIPS. Dado El paso intermedio que transforma los programas a Qprogramas, con este paso nos abstraemos de detalles de implementación de cada arquitectura. Esta extensibilidad amplía el alcance y la aplicabilidad de nuestra solución, permitiendo su uso en una variedad de plataformas hardware.

En conjunto, estas contribuciones representan un paso adelante en la mejora de la eficiencia, precisión y versatilidad de las herramientas disponibles para los desarrolladores de lenguajes ensambladores.

6.3. Limitaciones

A pesar de los avances y contribuciones significativas que hemos logrado con nuestra herramienta, es esencial reconocer algunas limitaciones inherentes a nuestro trabajo:

Cobertura Limitada de Instrucciones x86

Nuestra herramienta, en su estado actual, solo cubre un conjunto limitado de instrucciones x86. Aunque hemos incluido las instrucciones esenciales, hay una amplia gama de instrucciones x86 que no están contempladas en nuestra biblioteca. Esto puede limitar la aplicabilidad de la herramienta en programas más complejos que utilizan instrucciones avanzadas o específicas.

Análisis Interprocedural Ausente

Una de las limitaciones clave de nuestra herramienta es la falta de análisis interprocedural. Esto significa que nuestra herramienta está diseñada para trabajar solo en un único archivo de ensamblador a la vez. Si un programa se extiende a través de múltiples archivos de ensamblador o hace referencia a procedimientos en otros archivos, nuestra herramienta no podrá analizarlo de manera efectiva.

Foco en Archivos Individuales

Relacionado con la limitación anterior, nuestra herramienta está diseñada para trabajar con un solo archivo de ensamblador a la vez. Esto puede presentar desafíos cuando se trata de programas más grandes que están distribuidos en múltiples archivos.

Pérdida de la Estructura Original del Programa

Al transformar el programa original a un QProgram, se pierde la estructura original del programa. Esto significa que, aunque la herramienta puede señalar un error de tipo, no puede indicar la ubicación exacta del error en el código fuente original. Esto puede dificultar la tarea de corrección y depuración para los desarrolladores.

Reconocer estas limitaciones es esencial para guiar futuras investigaciones y desarrollos en este campo. Aunque hemos sentado una base sólida, queda claro que hay espacio para mejoras y expansiones para abordar estas limitaciones.

6.4. Trabajo Futuro

A partir de las contribuciones realizadas y las limitaciones identificadas, se vislumbran varias áreas de investigación y desarrollo para trabajos futuros:

Expansión de la Cobertura de Instrucciones x86

Dado que nuestra herramienta actual solo cubre un conjunto limitado de instrucciones x86, un área obvia de mejora sería expandir esta cobertura. Esto permitiría a la herramienta ser aplicable a una gama más amplia de programas y aumentar su utilidad en entornos de desarrollo reales.

Desarrollo de Análisis Interprocedural

Para superar la limitación del análisis de un solo archivo, se podría investigar y desarrollar capacidades interprocedurales. Esto permitiría a la herramienta analizar programas que se extienden a través de múltiples archivos de ensamblador y proporcionar una visión más holística del código.

Mejora en la Localización de Errores

Dado que la transformación a QProgram pierde la estructura original del programa, sería beneficioso desarrollar técnicas o métodos que permitan a la herramienta señalar no solo el error de tipo, sino también su ubicación exacta en el código fuente original.

Integración con Otros Compiladores y Arquitecturas

Dada la naturaleza extensible de la herramienta, el integraarla a otros compiladores y arquitecturas sería óptimo

Preservación de la Estructura del Programa

Investigar técnicas que permitan transformar el programa a un QProgram sin perder su estructura original podría ser esencial para mejorar la usabilidad y precisión de la herramienta.

Optimización y Eficiencia

A medida que la herramienta se expande para cubrir más instrucciones y archivos, será crucial garantizar que siga siendo eficiente en términos de tiempo y recursos.

Bibliografía

[1] . mypy webpage. `https://mypy-lang.org/`, 2023. último acceso Junio 26, 2023.

[2] . webpage documentaciónflow. `https://flow.org/en/docs/`, 2023. último acceso Junio 26, 2023.

[3] . *Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 1*. September 2016.

[4] E. Bonelli, A. Compagnoni, and R. Medel. Siftal: A typed assembly language for secure information flow analysis. 01 2004.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. 01 1976.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[7] R. H. R. J. Erich Gamma, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 3rd edition, 1994.

[8] G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[9] R. Monat, A. Ouadjaout, and A. Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In R. Hirschfeld and T. Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[10] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talx86: A realistic typed assembly language. 1999.

[11] Various. Sparta's framework repository, 2023. Accessed: July 20, 2023.

[12] F. K. B. C. D. Varro and S. Mcintosh. An empirical study of type-related defects in python projects. *IEEE Transactions on Software Engineering*, 48(2):3145–3158, 2021.