INSTITUTO TECNOLOGICO DE IZTAPALAPA I

INGENIERIA EN SISTEMAS COMPUTACIONALES

Reportes de Apuntes Semanales del

26 AL 30 DE ABRIL 2021

Presenta:

PEREZ ARMAS FAUSTO ISAAC

No. De control:

181080037

ASESOR INTERNO:

M.C. ABIEL TOMAS PARRA HERNANDEZ

CIUDAD DE MEXICO **JUNIO/2021**

INDICE

Actividades semana abril 26-30, 2021

Actividades individuales:

1) "Overview of Translation" del capítulo 1 del libro "Engineering a Compiler" de Cooper y Torczon (2012).

RESUMEN DE LA TRADUCCIÓN

Para traducir código escrito en un lenguaje de programación en código adecuado para ejecución en alguna máquina de destino, un compilador se ejecuta a través de muchos pasos. Para que este proceso abstracto sea más concreto, considere los pasos necesarios para generar código ejecutable para la siguiente expresión:

a ← a × 2 × b × c × d

donde a, b, c y d son variables, ← indica una asignación y × es el operador para la multiplicación.

En las siguientes subsecciones, rastrearemos la ruta que toma un compilador para convertir esta simple expresión en código ejecutable.

La interfaz

Antes de que el compilador pueda traducir una expresión en código de máquina de destino ejecutable, debe comprender tanto su forma o sintaxis como su significado, La interfaz determina si el código de entrada está bien formado, en términos de sintaxis y semántica. Si encuentra que el código es válido, crea una representación del código en la representación intermedia del compilador; Si no, informa al usuario con mensajes de error de diagnóstico para identificar el problema con el código.

Comprobación de sintaxis

Para comprobar la sintaxis del programa de entrada, el compilador debe comparar la estructura del programa contra una definición del lenguaje. Esto requiere una definición formal apropiada, un mecanismo eficiente para probar si o No la entrada cumple con esa definición, y un plan sobre cómo proceder en una entrada ilegal. Matemáticamente, el idioma de origen es un conjunto, generalmente infinito, de cadenas definido por un conjunto finito de reglas, llamado gramática. Dos pases separados en el front-end, llamado escáner y analizador, determina si o no el código de entrada es, de hecho, un miembro del conjunto de programas válidos definidos por la gramática.

Las gramáticas del lenguaje de programación generalmente se refieren a palabras basadas en sus partes. del habla, a veces llamadas categorías sintácticas. Basando las reglas gramaticales en partes del discurso permite que una sola regla describa muchas oraciones. Por ejemplo, en inglés, muchas oraciones tienen la forma

Oración → Sujeto verbo Objeto end Mark

donde el verbo y la marca final son partes del discurso, y Oración, Asunto y los objetos son variables sintácticas. La oración representa cualquier cadena con la forma descrito por esta regla. El símbolo "→" dice "deriva" y significa que una instancia de la derecha El lado de la mano se puede abstraer a la variable sintáctica en el lado izquierdo.

Considere una oración como "Los compiladores son objetos diseñados". El primer paso en la comprensión de la sintaxis de esta oración es identificar palabras distintas en el programa de entrada y para clasificar cada palabra con una parte del discurso. en un compilador, esta tarea recae en una pasada llamada escáner. El escáner lleva un escáner la pasada del compilador que convierte una cadena de personajes en un torrente de palabras Flujo de caracteres y lo convierte en un flujo de palabras clasificadas, que, es decir, pares de la forma (p, s), donde p es la parte del discurso de la palabra y s es su ortografía. Un escáner convertiría la oración de ejemplo en lo siguiente flujo de palabras clasificadas:

(sustantivo, "compiladores"), (ver o, "son"), (adjetivo, "diseñado"),

(sustantivo, "objetos"), (marca al final, ".")

En la práctica, la ortografía real de las palabras podría almacenarse en una tabla hash. y representado en pares con un índice entero para simplificar las pruebas de igualdad.

En el siguiente paso, el compilador intenta hacer coincidir el flujo de palabras categorizadas contra las reglas que especifican la sintaxis del idioma de entrada. Por ejemplo, un conocimiento práctico del inglés puede incluir los siguientes aspectos gramaticales reglas:

1 oración → Sujeto verbo Objeto endamar

2 asunto → sustantivo

3 sujeto → Modificador sustantivo

4 objeto → sustantivo

5 objeto → Modificador sustantivo

6 modificador → adjetivo

...

Mediante inspección, podemos descubrir la siguiente derivación para nuestro ejemplo oración: Oración de prototipo de regla- Oración

1 sujeto verbo Objeto endmark

2 sustantivo verbo Object endmark

5 sustantivo Verbo Modificador sustantivo endmark

6 sustantivo verbo adjetivo sustantivo endmark

La derivación comienza con la variable sintáctica Sentence. A cada paso, reescribe un término en la oración prototipo, reemplazando el término con un lado derecho que puede derivarse de esa regla. El primer paso usa la Regla 1 para reemplazar Sentencia. El segundo usa la Regla 2 para reemplazar el Asunto. El tercero reemplaza Objeto usando la Regla 5, mientras que el paso final reescribe Modificador con adjetivo de acuerdo con la Regla 6. En este punto, la oración prototipo generada por la derivación coincide con el flujo de palabras categorizadas producidas por el escáner.

La derivación prueba que la oración "Los compiladores son objetos diseñados". pertenece al lenguaje descrito por las Reglas 1 a 6. La oración es Gramaticalmente correcta. El proceso de encontrar derivaciones automáticamente es llamado análisis.

Una oración gramaticalmente correcta puede no tener sentido. Por ejemplo, la rica oración "Las rocas son vegetales verdes" tiene las mismas partes del discurso en el mismo orden que "Los compiladores son objetos diseñados", pero no tiene significado. Para comprender la diferencia entre estas dos oraciones se requiere conocimiento contextual sobre sistemas de software, rocas y vegetales. Los modelos semánticos que los compiladores utilizan para razonar sobre el lenguaje de programación. la pasada del compilador que comprueba la coherencia de tipos usos de nombres en el programa de entrada Los indicadores son más simples que los modelos necesarios para comprender el lenguaje natural. Un compilador crea modelos matemáticos que detectan tipos específicos de inconsistencia en un programa. Los compiladores comprueban la coherencia del tipo; por ejemplo, la expresión

a ← a × 2 × b × c × d

pueden estar bien formados sintácticamente, pero si byd son cadenas de caracteres, la oración aún podría no ser válida. Los compiladores también verifican la coherencia del número en situaciones específicas; por ejemplo, una referencia de matriz debe tener el mismo número de dimensiones que el rango declarado de la matriz y un procedimiento La llamada debe especificar el mismo número de argumentos que la definición del procedimiento. El capítulo 4 explora algunos de los problemas que surgen en el tipo basado en compilador comprobación y elaboración semántica.

Representaciones intermedias

El último problema que se maneja en la interfaz de un compilador es la generación de una forma ir del código. Los compiladores usan una variedad de diferentes tipos

de ir, según el idioma de origen, el idioma de destino y la traducción específica t0 ← a × 2

t1 ← t0 × b

t2 ← t1 × c

t3 ← t2 × d

a ← t3

Algunos irs representan el programa como un gráfico. Otros se asemejan a un programa de código ensamblador secuencial. El código en el margen muestra cómo podría verse nuestra expresión de ejemplo en un nivel bajo, ir secuencial El capítulo 5 presenta una descripción general de la variedad de tipos de IR que utilizan los compiladores.

Para cada construcción del lenguaje fuente, el compilador necesita una estrategia sobre cómo implementará esa construcción en la forma ir del código. Opciones específicas afectan la capacidad del compilador para transformar y mejorar el código. Por lo tanto, nosotros dedicamos dos capítulos a los problemas que surgen en la generación de ir para código fuente constructos. Los vínculos de procedimientos son, a la vez, una fuente de ineficiencia en el código final y el pegamento fundamental que une diferentes archivos fuente en una aplicación. Por lo tanto, dedicamos el capítulo 6 a las cuestiones que rodean llamadas a procedimiento. El capítulo 7 presenta estrategias de implementación para la mayoría de las demás construcciones de lenguaje de programación

El optimizador

Cuando el front-end emite ir para el programa de entrada, maneja las declaraciones uno a la vez, en el orden en que se encuentran. Así, la ir inicial El programa contiene estrategias generales de implementación que funcionarán en cualquier contexto circundante que el compilador podría generar. En tiempo de ejecución, el código se ejecutará en un contexto más restringido y predecible. El optimizador analiza la forma del código para descubrir hechos sobre ese contexto y usos ese conocimiento contextual para reescribir el código para que calcule el mismo responder de una manera más eficiente.

La eficiencia puede tener muchos significados. La noción clásica de optimización es para reducir el tiempo de ejecución de la aplicación. En otros contextos, el optimizador podría intentar reducir el tamaño del código compilado u otras propiedades como la energía que consume el procesador al evaluar el código. Todos estos las estrategias apuntan a la eficiencia. Volviendo a nuestro ejemplo, considérelo en el contexto que se muestra en la Figura 1.2a. La declaración ocurre dentro de un bucle. De los valores que utiliza, solo un y d cambio dentro del bucle. Los valores de 2, byc son invariantes en el círculo. Si el optimizador descubre este

hecho, puede reescribir el código como se muestra en Figura 1.2b. En esta versión, se ha reducido el número de multiplicaciones

de 4 · n a 2 · n + 2. Para n> 1, el bucle reescrito debería ejecutarse más rápido. Esto El tipo de optimización se analiza en los capítulos 8, 9 y 10.

Análisis

La mayoría de las optimizaciones consisten en un análisis y una transformación. El análisis determina dónde el compilador puede aplicar la técnica de forma segura y rentable. Análisis del flujo de datos Los compiladores utilizan varios tipos de análisis para respaldar las transformaciones. Datos una forma de razonamiento en tiempo de compilación sobre el flujo de valores en tiempo de ejecución razones de análisis de flujo, en tiempo de compilación, sobre el flujo de valores en tiempo de ejecución. Los analizadores de flujo de datos suelen resolver un sistema de ecuaciones de conjuntos simultáneos que se derivan de la estructura del código que se está traduciendo. Dependencia El análisis utiliza pruebas teóricas de números para razonar sobre los valores que se pueden

Transformación

Para mejorar el código, el compilador debe ir más allá de analizarlo. El compilador debe usar los resultados del análisis para reescribir el código en un formato más forma eficiente. Se han inventado innumerables transformaciones para mejorar los requisitos de tiempo o espacio del código ejecutable. Algunos, como descubrir cálculos invariantes de bucle y moverlos a ejecutados con menos frecuencia ubicaciones, mejorar el tiempo de ejecución del programa. Otros hacen el código Mas Compacto. Las transformaciones varían en su efecto, el alcance sobre el cual operan, y el análisis requerido para apoyarlos. La literatura sobre las transformaciones es rica; el tema es lo suficientemente grande y profundo para merecen uno o más libros separados. El capítulo 10 cubre el tema de escalar transformaciones, es decir, transformaciones destinadas a mejorar el rendimiento del código en un solo procesador. Presenta una taxonomía para organizar el tema y llena esa taxonomía con ejemplos.

El back-end

El back-end del compilador atraviesa la forma ir del código y emite código para la máquina de destino. Selecciona las operaciones de la máquina objetivo para implementar cada operación de ir. Elige un orden en el que se ejecutarán las operaciones eficientemente. Decide qué valores residirán en los registros y qué valores residirá en la memoria e inserta código para hacer cumplir esas decisiones

Actividades en equipo:

2) Multi-Level Intermediate Representation (MLIR)

- **MLIR**

Es un proyecto que define una representación intermedia (IR) común que unifica la infraestructura necesaria para ejecutar modelos de aprendizaje automático de alto rendimiento en Tensor Flow y en marcos de trabajo de AA similares.

Este proyecto incluye la aplicación de técnicas de HPC (computación de alto rendimiento), junto con la integración de algoritmos de búsqueda, como el aprendizaje por refuerzo. El objetivo de MLIR es reducir el costo para incorporar hardware nuevo y mejorar la usabilidad para los usuarios de Tensor Flow.

MLIR tiene algunas propiedades interesantes (forma SSA, operaciones de dialecto, regiones) que hacen que el trabajo sea mucho más fácil de cambiar. Pero lo que es más importante, el IR tiene un flujo de control explícito (CFG), que es importante para rastrear de dónde provienen las variables, pasan a través de todas las combinaciones de rutas.

Para inferir tipos y verificar la seguridad, esto es fundamental para asegurarse de que el código no pueda llegar a un estado desconocido a través de al menos una de las rutas posibles. Entonces, la razón principal por la que elegimos MLIR para ser nuestra representación es para que podamos hacer nuestra inferencia de tipo más fácilmente.
La segunda razón es que MLIR nos permite mezclar cualquier número de dialectos. Por lo tanto, podemos reducir el AST a una mezcla de dialecto de Verona y otros dialectos estándar, y los pases que solo pueden ver las operaciones de Verona ignorarán a los demás y viceversa.
También nos permite bajar parcialmente partes del dialecto a otros dialectos sin tener que convertir todo. Esto mantiene el código limpio (pasadas cortas y directas) y nos permite construir lentamente más información, sin tener que ejecutar una pasada de análisis enorme seguida de una pasada de transformación enorme, solo para perder información en el medio.
Un beneficio inesperado de MLIR fue que tiene soporte nativo para operaciones opacas, es decir. operaciones similares a llamadas a funciones que no necesitan definirse en ninguna parte.

MLIR no tiene una representación de cadena nativa y no hay una forma sensata de representar todos los tipos de cadenas en los tipos existentes.