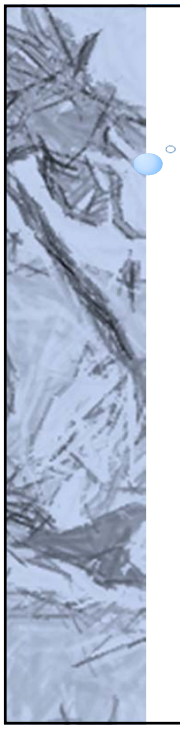
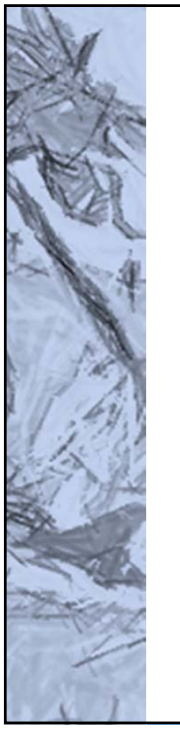


Tecnológico de Monterrey

Analizador Semántico

Diseño de Compiladores

Dr. Víctor de la Cueva
vcueva@itesm.mx



Análisis Semántico

- Es la fase del compilador que calcula la información adicional necesaria (para la compilación) una vez que se conoce la estructura sintáctica de un programa.
- Involucra el cálculo de información que rebasa las capacidades de las FCG.
- Como el análisis que realiza un compilador es estático por definición (tiene lugar antes de la ejecución), dicho análisis semántico también se conoce como Análisis Semántico Estático (ASE).

¿Qué verifica?

- Dependiendo del lenguaje de programación de que se trate puede verificar cosas como:
  - Que todos los identificadores estén declarados
  - Tipos
  - Que las relaciones de herencia tengan sentido
  - Que las clases sólo sean definidas una vez
  - Que las variables sólo estén declaradas una vez en el mismo *scope*
  - Que los métodos en las clases sean definidos sólo una vez
  - Que los identificadores reservados no estén mal usados
  - Y muchos otros

¿Qué involucra?

- En un lenguaje típico estáticamente tipado (e.g. C), el análisis semántico involucra:
  - La construcción de una ST para mantenerse al tanto de los significados de nombres establecidos en declaraciones, e
  - Inferir tipos y verificarlos en expresiones y sentencias con el fin de determinar su exactitud dentro de las reglas de tipos del lenguaje.

División

- El Análisis Semántico se puede dividir en dos categorías:
  - El análisis de un programa que requiere las reglas del lenguaje de programación para establecer su exactitud y garantizar una ejecución adecuada.
    - E.g. verificación de tipos estáticos
  - El análisis realizado por un compilador para mejorar la eficiencia de ejecución del programa traducido
    - Por lo general se incluye en el análisis de optimización o técnicas de mejoramiento de código
- Las dos categorías no son mutuamente exluyentes.

Herramientas

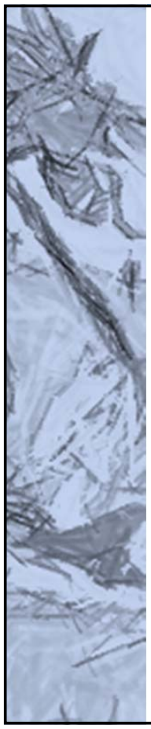
- El ASE involucra tanto la descripción de los análisis a realizar como la implementación de los análisis utilizando los algoritmos adecuados.
  - Es similar al léxico (RE y FA) o sintaxis (FCG y LL(1))
  - En ASE la situación no es tan clara:
    - En parte porque no hay un método estándar que permita especificar la semántica estática de un lenguaje, y
    - En parte porque la cantidad y categoría del ASE varía demasiado de un lenguaje a otro

Uno muy usado

- Un método para describir en análisis semántico que los escritores de compiladores usan muy a menudo con buenos efectos, es la identificación de atributos, o propiedades, de entidades del lenguaje que deben calcularse y escribir **ecuaciones de atributos** o **reglas semánticas**, que expresan la forma en la que el cálculo de tales atributos está relacionado con las reglas gramaticales del lenguaje.
- Un conjunto así de atributos y ecuaciones se denomina **gramática con atributos (GCA)**.
  - Las GCA son más útiles para los lenguajes que obedecen al principio de la semántica dirigida por la sintaxis

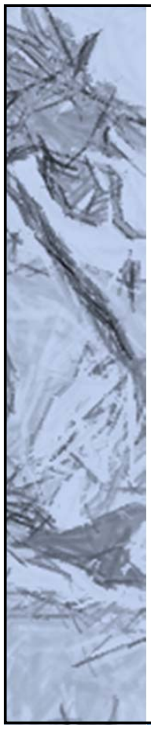
Semántica dirigida por sintaxis

- Asegura que el contenido semántico de un programa se encuentra estrechamente relacionado con su sintaxis.
- Todos los lenguajes modernos tienes esta propiedad.
- Desafortunadamente, el escritor de compiladores casi siempre debe construir una GCA a mano, a partir del manual del lenguaje, ya que rara vez la da el diseñador del lenguaje.



Algoritmos de implementación

- Los algoritmos para la implementación del análisis semántico tampoco son claramente expresables (como los algoritmos de análisis sintáctico).
- De nuevo, esto se debe a los mismos problemas respecto a la especificación del análisis semántico.
- Si el ASE se puede suspender hasta que todo el análisis sintáctico (y la construcción de un AST) esté completo, entonces la tarea de implementar el ASE se vuelve considerablemente más fácil.



ASE después del AST

- Consiste, en esencia, en la especificación de orden para un recorrido del AST, junto con los cálculos a realizar cada vez que se encuentra un nodo en el recorrido.
- Esto implica que el compilador debe ser de pasos múltiples.
- Afortunadamente, la práctica moderna permite cada vez más al escritor de compiladores utilizar pasos múltiples para simplificar los procesos de ASE y Generación de Código (CG).

Funciones principales del ASE

- Las dos áreas principales del ASE son:
  - Tabla de símbolos
  - Verificación de tipos

NOTA: No existe alguna herramienta de amplio uso para generar en forma automática los ASE.

Análisis Semántico

VERIFICACIÓN DE TIPOS (*TYPE CHECKING*)

Formalismo apropiado

- El formalismo apropiado en la verificación de tipos son las reglas lógicas de inferencia (*logical rules of inference*):
  - Tiene la forma:
    - **if** hipótesis == true, **then** conclusión == true
  - La verificación de tipos se hace vía razonamiento:
    - **if** E_1 y E_2 tienen cierto tipo, **then** E_3 tiene cierto tipo
  - Las reglas de inferencia son una notación compacta para estatutos **if-then**
    - Son estatutos de implicación donde alguna hipótesis implica alguna conclusión

Notación

- Su notación es fácil de leer con la práctica.
- Inicia con un sistema simplificado y gradualmente le agrega características.
- Bulding blocks:
  - El símbolo $\wedge$ es "and"
  - El símbolo $\Rightarrow$ es "if-then"
  - x:T significa "x tiene el tipo T"

AA 5

Formato

- Una regla de inferencia se escribe así:
  - Hipótesis$_1 \wedge \ldots \wedge$ Hipótesis$_n \Rightarrow$ Conclusión
- Por tradición, estas reglas se escriben como:
  - $\dfrac{\vdash Hipótesis_1 \ldots \vdash Hipótesis_n}{\vdash Conclusión}$
  - Donde $\vdash$ se lee, "se puede probar que…"
  - Significa, si se puede probar que la *hipótesis*$_1$ es verdadera y que , así sucesivamente, hasta la *hipótesis*$_n$ es verdadera, entonces se puede probar que la *conclusión* es verdadera.
- Las reglas de tipo de los lenguajes tienen hipótesis y una conclusión del tipo $\vdash e : T$

Reglas simples de tipos

$$\frac{i \text{ es una literal entera}}{\vdash i : Int}$$

[Int] regla para una constante entera

$$\frac{\vdash e_1 : Int \quad \vdash e_2 : Int}{\vdash e_1 + e_2 : Int}$$

[Add] regla para la suma

- Estas reglas son como *templates* que describen cómo tipar (poner tipos) enteros y expresiones de suma.
- Llenando los *templates* podemos producir un tipado completo para las expresiones.

AA 6

Condición de corrección (*correctness*)

- Una propiedad importante de cualquier sistema de tipos es que debe ser sano (*sound*).
- Un sistema de tipos es sound si:
  - Siempre que $\vdash e : T$
  - Entonces e se evalúa a un valor de tipo T
- Sólo queremos reglas que sean sound.
  - Pero algunas reglas sound son mejores que otras. AA 6

AST y las pruebas

- La verificación de tipos prueba hechos e:T
  - La prueba está en la estructura del AST
  - La prueba tiene la misma estructura que el AST
  - Se usa una regla de tipo por cada nodo del AST
- En la regla de tipo usada para el nodo e:
  - Las hipótesis son las pruebas de los tipos de las subexpresiones de e
  - La conclusión es el tipo de e
- Los tipos son calculados en una pasada *bottom-up* (posorden) del AST.
- Hay una correspondencia directa entre la estructura de una prueba y la forma del AST.

AA6.5

Regla para una variable

- Hasta ahora podemos definir de una forma muy directa, reglas de tipos razonables para cualquier constructor.
- Pero se presenta un problema: ¿cuál es el tipo de una referencia a una variable?
  - $\frac{x \text{ es una variable}}{\vdash x:?}$ [Var]
  - La regla no tiene suficiente información para dar el tipo de *x*.
- La solución es simple:
  - ¡Poner más información en la regla!

Ambientes de tipos (*type environments*)

- Un *type environment* proporciona los tipos para las variables libres (*free variables*).
  - Un *type environmente* es una función de identificadores (nombres de variables) a tipos
- Una variable es libre en una expresión si no está definida dentro de la expresión.

7.5

Función id → tipos

- Sea O una función de identificadores a tipos:
  - La sentencia $O \vdash e : T$ se lee:
    - Bajo la asunción de que las variables libres tienen los tipos dados por O, se puede probar que la expresión e tiene tipo T
  - Es decir, si me dices el tipo de las variables libres en una expresión, te puedo decir el tipo de la expresión
- A O se les conoce como ambiente de tipos.

AA 7.5

O como el ambiente

- Si la expresión e tiene variables libres, tenemos que ver la función O para que nos dé su tipo.
- Ahora nuestro problema con las variables libres se convierte en un problema sencillo y podemos escribir nuevas reglas:
  - $\frac{O(x)=T}{O \vdash x:T}$ $[Var]$
  - Para saber el tipo de x simplemente lo veo en mi ambiente de objetos.

Instrucción Let

- Podemos tener ahora una regla para la instrucción Let:
  - $\frac{O[T_0|x] \vdash e_1 : T_1}{O \vdash let\ x : T_0\ in\ e_1 : T_1}$
  - O[T|x] es la función O modificada en el único punto x para regresar T:
    - O[T|x](x) = T al aplicarle x a la función regresa T
    - O[T|x](y) = O(y) ya tenía definida a y
- Se puede observar que O se implementa con una ST.

¿Cómo se implementa?

- La función O se implementa como la tabla de símbolos:
  - Una o varias dependiendo de los bloques
  - Una tabla por cada bloque.
  - Si son varias se requerirá una pila de tablas de símbolos
- Tendremos una función recursiva llamada *typecheck* que toma dos argumentos:
  - Un *type environment* (ST) y una expresión
- El código se verá muy parecido a la lectura de la regla trasladada a código y esa es una de las principales ventajas de la notación de los *type systems*.

Ej función typecheck

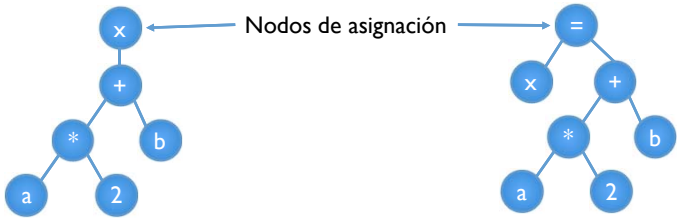
```
Typecheck(environment, e1+e2) {
    T1 = typecheck(environment, e1);
    T2 = typecheck(environment, e2);
    check T1 == T2 == Int;
    return Int; }

Typecheck(environment, let x:T ← e0 in e1) {
    T0 = typecheck(environment, e0);
    T1 = typecheck(environment.add(x:T), e1);
    check subtype(T0, T1);
    return T1; }
```

Recomendación en la implementación

- Recorre el árbol en posorden (de abajo hacia arriba), iniciando en la raíz.
- Verifica qué tipo de nodo están analizando:
  - Dependiendo del nodo, llama a la función `typecheck` o verifícalo directamente en el código usando la tabla de símbolos (o el stack de la tabla de símbolos), para cada uno de sus hijos.
  - Asigna el tipo encontrado al nodo analizado.

Ejemplo

- Para la expresión `x = a * 2 + b`, el AST es:

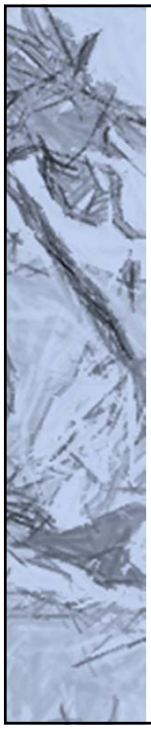


- Llama a `typecheck` desde la raíz, haciendo un recorrido en preorden (primero calcula el tipo de los hijos y luego define el de los padres).
- Inicia de abajo hacia arriba.

En resumen

- El ambiente de tipos proporciona los tipos para las variables libres.
- El ambiente de tipos se pasa como parámetro al AST de la raíz hacia las hojas.
- Los tipos son calculados en el AST de las hojas a la raíz.

Proyecto 3



Referencias

- A.V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Pearson (2012).
- K.C. Louden. *Contrucción de Compiladores: principios y práctica*. Thomson (2004).
- Alex Aiken. Compilers. Stanford Online (2018).
  - https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/about