Segundo Proyecto de Compilación

Título:

> Inferencia de tipos en el lenguaje de programación COOL, una aproximación a través de grafos.

Estudiantes:

- Alejandro Klever Clemente C-311
- Miguel Angel Gonzalez Calles C-311

1 Inferencia de Tipos

COOL es un lenguaje de programación estáticamente tipado, y aunque el lenguaje no presenta inferencia de tipos, esta es una característica muy útil que incorporaremos en un nuestro intérprete.

Nuestra solución a la inferencia de tipos se apoya en el uso básico de la teoría de grafos y en el uso del patrón de diseño visitor.

La inferencia de tipos de nuestro proyecto detecta para cada atributo, variable, parámetro de función o retorno de función el primer tipo que le puede ser asignado, modificando en el árbol de sintaxis abstracta el string `AUTO_TYPE` por el nombre del tipo correspondiente y asignando los tipos correspondientes en el contexto y el ámbito en que seon declarados.

1.1 Algoritmo y Grafo de Dependecias

Entrada : Un árbol de sintaxis abstracta, un contexto con todos los tipos declarados en el programa de COOL.

Salida : Un árbol de Sintaxis Abstracta, Un Contexto y un Scope con tipos bien tagueados.

Algoritmo : Durante el recorrido del AST será construido un grafo dirigido cuyos nodos encerrarán el concepto de las expresiones marcadas como `AUTO_TYPE` y las aristas representan las dependencias entre las expresiones de estos nodos para inferir su tipo. Sea `E1` una expresión cuyo tipo estático es marcado como `AUTO_TYPE`, y sea `E2` una expresión a partir de a cual se puede inferir el tipo de estático de `E1` entonces en el grafo existirá la arista `<E2, E1>`. Una vez construido el árbol se comenzará una cadena de expansión de tipos estáticos de la forma `E1, E2, ..., En` donde `Ej` se infiere de `Ei` con `1 < j = i + 1 <= n` y `E1` es una expresión con tipo estático definido, al cual llamaremos átomo. Cuando todos los átomos se hayan propagado a través del grafo los nodos que no hayan podido ser resueltos serán marcados como tipos `Object` al ser esta la clase mas general del lenguaje.

Implementación : Para esto creamos una estructura llamada `DependencyGraph` donde podemos crear nodos como una estructura llamada `DependencyNode` y arcos entre ellos. La estructura `DependencyGraph` consiste en un `OrderedDict` de nodos contra lista de adyacencia. Esta lista de adyacencia contiene los nodos a los cuales la llave propagar su tipo, estos nodos de la lista tienen un orden y esto es fundamental para el algoritmo de solución de inferencia. Si tenemos un caso `{x: [y, z]}` donde `x`, `y`, `z` son nodos, entonces el algoritmo determinará el tipo de `y` y `z` antes de continuar con todas sus cadenas de expansión, por lo que si `z` forma parte de una cadena de expansión de `y` entonces `y` no propagará su tipo a `z` ya que `x` lo hizo antes. Como se puede ver el algoritmo es un BFS simple donde en la cola, al inicio, serán incluido los nodos del grafo que tengan su tipo definido, es decir que no sea `AUTO_TYPE`.

1.2 Nodos de Dependencia

Cada nodo del grafo será una abstracción de un concepto en el que se use un tagueo explícito de `AUTO_TYPE` y tendrá las referencias a las partes del proceso de semántica del programa, además de que cada nodo contará con un método `update(type)` el cual actualiza el tipo estático de estos conceptos.

```
class DependencyNode:
    pass

class AtomNode(DependencyNode):
    """Nodo base el cual es creado a partir de expresiones que
    contienen tipo estático u operaciones aritméticas"""
    pass

class AttributeNode(DependencyNode):
    """Atributo de una clase"""
    pass

class ParameterNode(DependencyNode):
    """Parámetro de una función"""
    pass

class ReturnTypeNode(DependencyNode):
    """Tipo de retorno de una función"""
    pass

class VariableInfoNode(DependencyNode):
    """Variables declaradas en el scope."""
    pass
```

1.3 Casos factibles

El algoritmo funciona de manera análoga para atributos, variables, parámetros y retorno de funciones. Explicado de forma recursiva puede ser visto como:

- Un `AUTO_TYPE` será sustituido por su tipo correspodiente si este forma parte de una operacion que permita saber su tipo, o es usado en una expresión de la cuál es posible determinar su tipo.
- Es importante señalar en que contexto estas dependencias son tomadas en cuenta:
    - Para los atributos marcados como `AUTO_TYPE` su tipo podrá ser determinado dentro del cuerpo de cualquiera de las funciones de la clase, o si es detectable el tipo de la expresión de inicialización.
    - Para las variables su tipo será determinado dentro del scope donde estas son válidas.
- Para los parámetros su tipo será determinado dentro del cuerpo de la función o cuando esta función sea llamada a través de una operacion de dispatch.
- Para los retornos de funciones, su tipo será determinado con su expresión y los llamados a dicha función a través de una operacion de dispatch.
- En las expresiones if-then-else o case-of asignan automáticamente el tipo `Object` si al menos una de sus ramificaciones devuelve una expresión que no tiene tipo definido (`AUTO_TYPE`), en caso contrario asignará el `join` de las ramificaciones.

1.3.1 Ejemplos de casos factibles para la inferencia

En este caso la expresión `d + 1` desambigua a `d` en un `Int` y luego `c` se infiere de `d` , `b` se infiere de `c` , `a` se infiere de `b` y el retorno de la función de infiere de `a` . Quedando todos los parámetros y el retorno de la función marcados como `Int` .

```
class Main {
    function(a: AUTO_TYPE, b: AUTO_TYPE, c: AUTO_TYPE, d: AUTO_TYPE): AUTO_TYPE {
        {
            a <- b;
            b <- c;
            c <- d;
            d <- a;
            d + 1;
            a;
        }
    };
}
```

Similar al caso anterior pero en esta ocasión incluyendo atributos, la expresión los `x + y` infiere a los parámetros `x` y `y` como `Int` y tambien al atributo `b`, `a` se infiere de `b`. El tipo de retorno de `create_point()` se infiere de su porpia cuerpo con la expression `new Point` y esta a su vez infiere el tipo de retorno de `init()`.

```
class Point {
    a: AUTO_TYPE;
    b: AUTO_TYPE;

    init(x: AUTO_TYPE, y: AUTO_TYPE): AUTO_TYPE {{
        a <- b;
        b <- x + y;
        create_point();
    }};

    create_point(): AUTO_TYPE { new Point };
}
```

Probando con funciones recursivas tenemos el caso de fibonacci tipo de `n` se infiere al ser usado en las expresiones `n <= 2`, `n - 1`, `n - 2`, las cuales lo marcan como `Int`, `fibonacci(n - 1) + fibonacci(n - 2)` marca al retorno de la función como `Int` y la expresión if-then-else lo marca como `Object`. En este último caso la expresión `fibonacci(n - 1) + fibonacci(n - 2)` termina de analizarse primero que la expresión if-then-else por lo cual el tipo de retorno será `Int` el cual fue el primero que se definido, lo cual demuestra que el orden en el que se analizan las inferencias importan.

```
class Main {
    fibonacci(n: AUTO_TYPE): AUTO_TYPE {
        if n <= 2 then 1 else fibonacci(n - 1) + fibonacci(n - 2) fi
    };
}
```

El caso de Ackerman es bastante interesante, en nuestro algoritmo importa el orden en el que fueron definidas las dependencias. `m` y `n` son inferibles como `Int` a partir de las expresiones `n + 1` y `m - 1` respectivamente, y el tipo de retorno de `ackermann` en inferible por `n` al ser usado como segundo parámetro en un llamado de si mismo, por lo cual será `Int`. La influencia de la expresión `if-then-else` se ve anulada por el orden de inferencia.

```
class Main {

    ...

    ackermann(m: AUTO_TYPE, n: AUTO_TYPE): AUTO_TYPE {
        if m = 0 then n + 1 else
            if n = 0 then ackermann(m - 1, 1) else
                ackermann(m - 1, ackermann(m, n - 1))
            fi
        fi
    };

    ...

}
```

1.4 Casos No factibles

No es posible determinar el tipo de una variable, atributo, parámetro, o retorno de función si para este se cumple el caso general y su caso especifico correspondiente.

1.4.1 Casos generales

El tipo es utilizado en expresiones que no permiten determinar su tipo, o solo se logra determinar que poseen el mismo tipo que otras variables, atributos, parámetros o retorno de funciones de las cuales tampoco se puede determinar el mismo.

```
class Main {
    b: AUTO_TYPE;
    c: AUTO_TYPE;

    ...

    f(a: AUTO_TYPE, d: AUTO_TYPE): AUTO_TYPE {
        {
            b <- a;
            d <- c;
            f(a);
        }
    };

    factorial(n: AUTO_TYPE): AUTO_TYPE {
        if n = 1 then 1 else n * factorial(n - 1) fi
    };
}
```

En este ejemplo solo es posible inferir el typo del parámetro `n` de la función `factorial`, su tipo de retorno, el parámetro `a` de la función `f`, su tipo de retorno y el atributo `b` de la clase `Main`, el resto será marcado como `Object`.

```
class Main {
    main (): Object {
        0
    };

    f(a: AUTO_TYPE, b: AUTO_TYPE): AUTO_TYPE {
        if a = 1 then b else
            g(a + 1, b / 1)
        fi
    };

    g(a: AUTO_TYPE, b: AUTO_TYPE): AUTO_TYPE {
        if b = 1 then a else
            f(a / 2, b + 1)
        fi
    };
}
```

En este último caso es posible determinar el tipo de los parámetros `a` y `b` de ambas funciones `f` y `g` ya que estos participan en operaciones aritmétcas, sin embargo los tipos de retorno de estas funciones no son determinables ya que no se usan en otro contexto en un llamado en una de las ramas de la clausula `if-then-else`. Como nuestro algoritmo no propaga los tipos definidos entre ramas de expresiones como los `if-then-else` o `case-of` entonces el tipo de retorno no puede ser definido, luego por defecto se etiquetan como `Object`.

1.4.2 Casos Particulares

Para variables: si estas no son utilizadas en el ámbito en que son marcadas como `Object`.

```
class Main inherits IO {

    ...

    f(): Int {
        let a: AUTOTYPE, b: AUTO_TYPE in {
            1;
        }
    };

    ...

}
```

Para parámetros: si dentro del cuerpo de la función estas no son utilizadas y no existe otra función que llame a esta con argumentos con tipado estático definidos, serán marcadas como `Object`:

```
class Main inherits IO {

    ...

    f(a: AUTO_TYPE): Int{
        1
    };

    ...

}
```

Para atributos: si no es posible determinar el tipo de la expresión de inicializacion o si dentro del cuerpo de todas las funciones de su clase correspondiente este no son utilizadas, serán marcadas como `Objects`.

```
class Main inherits IO {
    b: AUTO_TYPE;
    c: AUTO_TYPE;

    ...

    f(a: AUTO_TYPE): AUTO_TYPE{
        if a < 3 then 1 else f(a - 1) + f(a - 2) fi
    };
}
```

Para el retorno de funciones: si no es posible determinar el tipo de su expresión.

```
class Main inherits IO {

    ...

    f(a: Int): AUTO_TYPE{
        if a < 3 then a else f(a - 3) fi
    };
}
```

1.5 Expresiones atómicas

- Valores constantes (ej: 2, "hello", true).
- Operaciones aritméticas, las cuales influyen en sus operandos que sean `AUTO_TYPE`.
- Operaciones lógicas, las cuales influyen en sus operandos que sean `AUTO_TYPE`.
- Llamdos a funciones con valor de retorno conocido.
- Instancias de clases.
- Variables de las cuales se conoce su tipo.
- Bloques donde se puede determinar el tipo de la última expresión.

2 Lexing y Parsing

Para el proceso de lexing y parsing usamos el paquete de Python PyJapt, cuyos autores coinciden con los de este proyecto.

3 Proceso de Semántica

El proceso de semantica es bastante similar al visto en clases prácticas de la 12 a la 15 con algunas pequeñas modificaciones:

- Recoleccion de tipos.
- Construccion los métodos y atributos de los tipos.
- Comprobamos que no existan dependencias cíclicas con un ordenamiento topológico de los tipos en su árbol de jerarquía
- Chequeo del la sobreescritura de métodos.
- Inferencia de tipos.
- Chequeo de tipos.
- Ejecución del programa.

Cada uno de estos procesos hace uso del patrón visitor visto en clases prácticas para recorrer el ast y realizar el analisis de cada uno de los procesos.

3.1 Ejecución de código

Para la ejecución hemos creado la abstracción de una instancia de un objeto. Con esto conseguimos en todo momento saber sobre que objecto en memoria estamos ejecutando un método y también tenemos acceso a los valores particulares de sus atributos en todo momento del programa. Para guardar el orden de ejcución de las funciones llevaremos una cola donde cada vez que ocurra un llamado a función colocaremos en el tope de la pila la instancia actual en la que estamos y tomará el control de la ejecución la instancia de la cual realizamos el dispatch. Finalmente cuando termine la ejecución de una función retomará el control del programa la

instancia en el tope de la pila. El proceso se retira hasta que la pila esta vacía. Para comenzar ejecución de cualquier programa en `COOL` basta con comenzar con la instrucción `(new Main).main()`.

4 CLI-API

Para la cómoda utilizacion del intérprete hemos usado el paquete de python `typer` para crear una api-cli bastante sencilla, basta con ejecutar el comando `python cool --help` y obtendrá como salida lo siguiente:

```
Usage: cool [OPTIONS] COMMAND [ARGS]...

Options:
--install-completion  Install completion for the current shell.
--show-completion     Show completion for the current shell, to copy it or
                        customize the installation.

--help                Show this message and exit.

Commands:
infer
run
serialize
```

Se se puede apreciar existen 3 comandos principales:

- `infer` el cual recibe un archivo .cl con un programa en COOL con tipos `AUTO_TYPE` y devuelve un programa en COOL con todos los `AUTO_TYPE` reemplazados por sus tipos correspondientes.
- `run` el cual recibe como entrada un archivo .cl con un programa en COOL y ejecuta dicho programa.
- `serialize` el cual vuelve a generar y serializar el parser y el lexer del lenguaje tras hacer modificaciones en su gramática. (Pensado para los desarrolladores)
- En ambos casos se tiene como parámetro adicional el `--verbose` para observar los distintos procesos por los que pasa el proceso de compilación.

5 Testing

En la carpeta `tests` se encuentra las carpetas `execution`, `inference`, `lexer`, `parser`, `semantic` las cuales contienen casos de pruebas. Para correr todas las pruebas basta con hacer `pytest tests` y se probaran todos los casos menos los de la carpeta `execution`, la cual contiene programa en cool que deberan ser ejecutados a mano usando la cli-api de nuestro intérprete.