Segundo Proyecto de Compilación
Inferencia de tipos para COOL

Amalia Ibarra Rodríguez AMALIA.IBARRA@ESTUDIANTES.MATCOM.UH.CU
grupo 312
Gabriela B. Martínez Giraldo GABRIELA.MARTINEZ@ESTUDIANTES.MATCOM.UH.CU
grupo 312

Tutor(es): Lic. Juan Pablo Consuegra

1. Introducción

Cool es un lenguaje de programación orientado a objetos que, aunque pequeño, tiene muchas características relevantes de lenguajes modernos como son lo objetos, el tipado dinámico y otros. En el presente trabajo se presentan nuestros esfuerzos en función de efectuar inferencia de tipo para Cool. Trataremos de explicar con el

2. Desarrollo

2.1 Gramática

Este primer acercamiento al lenguaje Cool requirió, como cualquier otro lenguaje, la definición de una gramática. En el archivo cool_grammar.py puede observarse como se consideró la misma para no aleajarnos de lo que referí el manual.

Como ahí se puede observar un porgrama de Cool consiste en una serie de definiciones de clases. Cada clase a su vez posee un conjunto de atributos y de funciones. Las expresiones que pueden formar parte de dichas funciones son el corazón del lenguaje.

No es nuestra intención hacer la historia muy larga, pues para eso está el manual, pero sí considerábamos necesario hacer pequeñas aclaraciones de algunos niveles en que se dividió dicha grámatica. En la imagen 1 se pueden apreciar varios niveles intermedios de esta:

1. <comp>, que representa las operaciones donde se comparan elementos.
2. <arith>, que representa la operaciones de uma y resta.
3. <term>, para la multiplicación y división.
4. <factor>, como representación de los operadores unarios isvoid, opuesto y new.
5. <element> para las expresiones entre paréntsis, los *block* y las llamadas a funciones.
6. <atom> como el nivel más básico, donde se excuentran los números, ids, las expresiones booleanas y los string.

Estos niveles fueron orientados de esta forma en función de lo que el manual de Cool refería en cuanto a la precedencia de los operadores.

```
<comp> := <comp> < <arith>
<comp> := <comp> = <arith>
<comp> := <comp> <= <arith>
<comp> := <arith>
<arith> := <arith> + <term>
<arith> := <arith> - <term>
<arith> := <term>
<term> := <term> * <factor>
<term> := <term> / <factor>
<term> := <factor>
<factor> := isvoid <element>
<factor> := ~ <element>
<factor> := new id
<factor> := <element>
<element> := ( <expr> )
<element> := { <block> }
<element> := <element> . <func-call>
<element> := <element> @ id . <func-call>
<element> := <func-call>
<element> := <atom>
<atom> := int
<atom> := id
<atom> := true
<atom> := false
<atom> := string
```

Figura 1: Fragmento de la gramática de Cool.

2.2 Tokenizer y Parser

Para tokenizar la entrada se utilizó una herramienta bastante práctica: *PLY*. Esta no es más que una impementación en python de las herramientas de parsing lex y yacc. Mediante el módulo *lex* nos provee de un analizador léxico ya implementado.

Para utilizarlo se definieron una serie de reglas para que el tokenizador trabajara en las cadenas de entrada y nos devolviera los tokens a gusto. Para ello se trabajó fundamentalmente con el módulo *re* de python que permite definir expresiones regulares. En el archivo *token_rules* se pueden observar dichas reglas.

El parse de *ycc* en cambio no fue el que utilizamos;

decidimos seguir con nuestra implementación de proyectos pasados, la cual se puede apreciar en el archivo *shift_reduce_parsers*.

2.3 Inferencia de Tipos

Para la inferencia de tipos se implementaron 3 recorridos por el AST auxiliándonos del patrón `visitor` visto en clases. El primero de estos se encuentra en el archivo src/tset_builder.py y tiene como objetivo construir una estructura que permita almacenar para cada variable el conjunto de tipos que esta puede tener. Esta estructura (a la que llamaremos `Tset`) es similar al `Scope`, de hecho nuestra primera intención fue construirla a partir del mismo, pero como los hijos de un scope no se sabe a qué nodo pertenecen, tuvimos que descartar esta idea, pues para las siguientes etapas del proceso esta información es imprescindible.

```
class Tset:
    def __init__(self, parent=None):
        self.locals = {}
        self.parent = parent
        self.children = {}

    def create_child(self, node):
        child = Tset(self)
        self.children[node] = child
        return child
```

Analicemos las propiedades de `TSet`:

1. `locals` diccionario que guarda dado el nombre de una variable el conjunto tipos que esta puede tener.
2. `children` diccionario que dado un nodo devuelve el `Tset` correspondiente.
3. `parent` referencia al `Tset` padre.

Cómo se guardan las variables y qué tipos se le asignan? Sencillo, se visitan los nodos donde se declaran variables (`AttrDeclarationNode`, `FuncDeclarationNode`, `LetNode`, `CaseNode`) y se le asigna a cada una el conjunto que contiene el tipo declarado, si el mismo es `AUTO_TYPE` el conjunto que le corresponde es el que contiene todos los tipos existentes.
El segundo recorrido se encarga de reducir los conjuntos de tipos asociados a las variables y se ejecuta mientras existan cambios entre los tipos devueltos en la última pasada y la actual. A continuación se muestra el código para más claridad.

Toda visita a un nodo de una expresión retorna un conjunto de tipos. En qué situaciones se puede reducir el conjunto de tipos asociado a una variable?

1. Declaración de atributos y Asignación

```
@visitor.when(ProgramNode)
def visit(self, node, tset):
    backup_tset = Tset()
    while not backup_tset.compare(tset):
        backup_tset = tset.clone()

        for declaration in node.declarations:
            self.visit(declaration, tset.children[declaration])

    tset.clean()
    return tset
```

```
class A {
 main(b:AUTO_TYPE) : Int {
   b <- "example"
 }
}
```

En toda asignación podemos reducir los tipos de la variable de la izquierda con los de la expresión que se le está asignando, en este caso podemos reducir el tipo de la variable b a String. El caso de la inicialización de atributos es análogo.

2. Operaciones aritméticas y de comparación
   Toda expresión que se utilice como operando en una operación aritmética o comparación debe ser de tipo Int.
3. Condicionales en expresiones if y ciclos deben ser de tipo Bool
4. Llamados a función

```
class A {
 b : AUTO_TYPE;
 main() : Int {
   b.solve()
 }
}
```

En el caso anterior se pueden reducir los tipos de `b` a los tipos que contienen un método con el mismo nombre y cantidad de parámetros que `solve()`.

Para reducir los tipos se utiliza la función `reduce_set()`. Esta halla la intersección entre los conjuntos de entrada y si contiene elementos la devuelve, en caso contrario devuelve la unión de ambos conjuntos con la adición del tipo `InferenceError`. Cuando la intersección entre estos dos conjuntos es vacía estamos ante un posible error en el código, de ahí que se le añada `InferenceError` para marcar que a partir de ahora no vamos a intersectar este conjunto con más ninguno, siempre vamos a hallar la unión. Decimos "posible" error en el código por situaciones como las siguientes:

```
class A {
...
```

```
}
class B inherits A {
...
}

class C {
 main() : Int {
   a <- new B ;
   a <- new A;
 }
}
```

Luego de la primera asignación la variable queda con tipo B, por tanto al hacer la segunda intersección esta va a ser vacía, sin embargo esto no es un error ya que B hereda de A, luego la variable c tiene tipo A. Es por esto que en la tercera pasada preferimos asignarle a cada variable el ancestro común de todos los tipos que quedan en su lista, en vez de lanzar error cada vez que estemos ante una intersección vacía.

Una vez reducidos los tipos es necesario modificar el `AST`. Para esto se hace un tercer recorrido por el `AST`, cuya implementación se encuentra en `src/tset_merger.py`. Como mencionamos en el párrafo anterior este recorrido se encarga de asignarle a cada variable el ancestro común a todos los tipos que quedaron en su lista.

Las variables que antes tenían tipo `AUTO_TYPE` ahora cambiaron su tipo, por lo que hace falta hacer otra pasada chequeando tipos, para esto repetimos las 3 pasadas vistas en clases: `TypeCollector`, `TypeBuilder` y `TypeChecker`.

3. Aplicación visual

Para poder apreciar mejor el trabajo realizado se implementó una pequeña aplicación visual usando streamlit. Para inciar el proyecto debe correr strealit run main.py en la consola y abrir navegador.

Una vez ahí debe insertar el código que desee analizar, y seleccionar compilar. Puede seleccionar entre tre opciones para no sobrecagar la página.