REPORTE

Rodrigo Pino C412

Adrián Portales C412

Semántica e Inferencia

Nuestro proyecto hace uso de `AUTO_TYPE` con la que incorpora inferencia de tipos al lenguage Cool. La inferencia se realiza varias en distintos vistores. En nuestra implementación debido a que le inferencia se apoya fuertemente sobre el reglas semánticas, el chequeo semántico se realiza a la par que la inferencia, y dividido de igual manera por los visitores.

La idea principal para realizar la inferencia es considerar todo declaración como `AUTO_TYPE` no como un tipo específico sino como un conjunto de tipos, donde inicialmente ese conjunto esta compuesto por todos los tipos definidos en un programa Cool. Los tipos declarados específicamente como `Int` o `String` se consideran conjuntos con un solo elemento.

En Cool muchas veces las expresiones se ven obligadas a conformarse a un tipo definido por el usuario. Deben corresponder con el tipo definido de una variable, argumento o retorno de una función. También debe obedecer las reglas semánticas, cuando están presente frente a una operación aritmética, o en una posición donde se espera que el resultado sea `Bool`. Para reducir los conjuntos de tipos en presencia de `AUTO_TYPE` realizamos lo siguiente:

1. Cuando el tipo declarado de una variable esta bien definido (diferente de `AUTO_TYPE`) , se eliminan del conjunto de tipos inferidos de la expresión los elementos que no conforman a dicho tipo bien definido.
2. Cuando el tipo declarado de una variable es `AUTO_TYPE`, esta se puede reducir analizando que valores debe tener para conformarse con los tipos de la expresión inferida.
3. Cuando ambos tipos, tanto el definido como el inferido son `AUTO_TYPES` se busca que valores puede tener el segundo para conformarse al primero, y que valores el primero para que el segundo se conforme.

Para tratar cada caso el inferenciador se divide en tres partes:

1. **soft inferencer** que aplica la primera regla y tercera regla. Se le llama **soft** porque perdona y no realiza ningún tipo de chequeo semántico y permite cosas como que un conjunto tengo dentro de si dos tipos sin un ancestro común.
2. **hard inferencer** aplica la primera y la tercera regla, y fuerza el chequeo semántico sobre todas las expresiones. No permite tipos sin ancestros comunes dentro de un mismo conjunto.
3. **back inferencer** aplica la segunda y tercera regla. Dada las expresiones trata de reducir los conjuntos de los tipos definidos como `AUTO_TYPE` (e.g. parámetros de una función, retorno de una función, o declaracion de variables)
4. **types inferencer** reduce todos los conjuntos de tipos de cada nodo al mayor ancestro en todos los casos, excepto cuando se trata del valor de retorno de una función, en el que reduce al ancestro común más cercano de los tipos del conjunto.

Cada inferenciador se ejecuta secuencialmente, una sola vez, exceptuando por el **back inferencer** que puede ejecutarse tantas veces como sea necesario.

Soft Inferencer

El **soft inferencer** es permisivo pues como es el primero en leer el programa, puede que un conjunto de tipos inválidos en la línea 5 se vuelva válido más adelante en el código.

En este ejemplo no funcional (Lanza `RuntimeError` debido a que `a` es `Void`) donde el tipo de `a` puede ser cualquiera, al leer `a.f()`, se reducen los tipos de a de {`Object`, `String`, `IO`, `Int`, `Bool`, `Main`, `A`, `B`} a tan solo {`A`, `B`}. No obstante `A` y `B` no tienen un ancestro común dentro del conjunto, luego `a` posee un tipo invalido.

Luego cuando se lee `a.g()` el conjunto de tipos se reduce a solo {`A`}.

```
class Main {
    a : AUTO_TYPE;
    method main():AUTO_TYPE {
        {
            a.f();  // Boom si no es el soft inferencer
            a.g();  // Solucionado
        }
    }
}
class A {
    method f():Int{
        3 + 3
    }
    metod g():String{
        "yisus"
    }

};
class B {
    method f():String{
        "3 + 3"
    }
};
```

SELF_TYPE

La combinación de `SELF_TYPE` con `AUTO_TYPE` trajo sus problemas, sobre todo porque funciona como un comodín que puede tener un tipo dependiendo de las circunstancia. Logramos una mejor integración entre estos fue posible intercambiando el `SELF_TYPE` por la clase donde se encuentra analizando en ese momento.

Generación de Código Intermedio

Para producir código CIL se toma como principal el guía el capitulo 7 del libro de `Cool Compilers` por su simpleza y facilidad luego para traducirlo a smips.

El programa original se divide en tres secciones:

- En **types** se guarda la signatura de los tipos. Nombre de atributos y funciones.
- **data** almacena todos los `String` definidos en tiempo de compilación por el usarion así como `Strings` definidos durante la propia generación de código.
- En **code** se encuentra el equivalente en CIL de las funciones definidas en Cool. Cada función en vez de tener expresiones y sub-expresiones complejas tienen una secuencia de operaciones más sencillas que producen un resultado equivalente.

Types

Contiene solo el nombre de la clase, los métodos y su identificador único para buscarlo cuando se necesite llamar a un método y los atributos de la misma. Los tipos también contienen dentro de ellos los atributos de las clases que heredan al igual que sus funciones.

Para todas las clases se les genera una función `init` donde se inicializan los valores iniciales de todos sus atributos. Si el atributo no esta inicializado, se inicializa por defecto apuntando a la cadena vacía en caso de ser de tipo `String` o con valor 0 en otro caso.

En caso de que el atributo se inicialice con una expresión, se transforma a operaciones en CIL y se le asigna el resultado final al atributo correspondiente.

La función `init` se ejecuta siempre que se instancie una clase.

Data

Se almacenan todos las cadenas definidos por el usuario en el programa Cool. Ademas se tiene también la cadena vacía a la cual apuntan las variables `String` sin inicializar. Durante la generación de código se almacena aquí además los mensajes de errores en ejecución.

Code

Cada expresión de Cool tiene una representación en secuencia de operaciones en CIL. Se asegura siempre que dentro de esa secuencia haya una instrucción que guarde en una variable local el resultado final de dicha expresión.

Las expresiones no siempre tienen la misma secuencia de instrucciones, pues necesitan muchas veces del valor de sus sub-expresiones. El workflow para producir una serie de operaciones para una expresión es:

1. Produce las operaciones de todas sus sub-expresiones
2. Produce las operaciones propias, sustituyendo donde se necesite cierta sub-expresion por la variable local donde esta guardada su resultado final.
3. Organiza las operaciones, crea una variable local donde se almacene el valor final propio y retorna

Existen ciertas expresiones que en CIL se pueden reducir hasta un punto y no mas, como la igualdad entre dos variables de tipo `String`, o como obtener un substring.

Existen otras que no es necesario que lleguen a smips como el operador unario `isVoid`. Como en smips todo son enteros, se puede saber dado el tipo estático si tiene sentido calcularlo. Para una variable de tipo `Int`, `String` o `Bool`, `isVoid` siempre retorna falso, en cambio con los demás tipos se evalúa la dirección de memoria, si esta es 0 (Equivalente a `Void` en nuestra implementación) el resultado de la expresión es `true` o `1` sino es `false` o `0`.

Durante la generación de código se genera también las excepciones que pueden ser lanzadas durante la ejecución:

- División por cero
- El despacho ocurre desde un tipo sin inicializar (`Void`)
- El rango del substring no es válido
- Ninguna rama de algún `case of` es igual al tipo de la expresión

Es posible para el usuario definir variables con mismos nombres con distintos contextos, para tratar con esto se reutilizan una versión simplificada del `Scope` de la semántica, donde se almacenan según el contexto la variable definida por el usuario y su traducción a Cool. Gracias a esto, en el ejemplo siguiente se conoce siempre a que variable `x` se refiere el programa:

```
# COOL
let x:int = 3
    in (let x:int = 4 in x) + x
# CIL
local let_x_0
local let_x_1
...
```

Transformaciones

Ejemplos de traducción de Cool a CIL

Declaración de Clase

Cool Input

```
class C {
    -- Initialized attributes
    a1: <attr_type> <- <expression>;
    a2: <attr_type> <- <expression>;
    ...
    am: <attr_type> <- <expression>;

    -- Functions
    f1(<param_list>) { <expression> }
    f2(<param_list>) { <expression> }
    ...
    fn(<param_list>) { <expression> }

}
```

CCIL Output

```
type C {
    attribute a1;
    ...
    attribute am;

    method f1 : <func_code_name>;
    ...
    method fn : <func_code_name>;
}
```

Herencia de Clases

Cool Input

```
class A {
    a1:<attr_type>
    f1():{...}
}

class B inherits A {
    b1:<attr_type>
    g1():{...}
}
```

CCIL Output

```
type A {
    attr a1;
    method f1 : f_f1_A
}

type B {
    attr a1;
    attr b1;
    method f1: f_f1_A
    method g1: f_g1_B
}
```

While Loop

Cool Input

```
while (<cond_expr>) loop <expr> pool
```

CCIL Output

```
label while_init
x = <resultado_numerico_de_cond_expr>
ifFalse x goto while_end

<do_body_expr>

goto while_init
label while_end
```

If Then Else

Cool Input

```
if <if_expr> then <then_expr> else <else_expr> fi
```

CCIL Output

```
<do_if_cond_expr> # Produce todas las operaciones de la expr de la cond. inicial
x = <if_cond_expr_result>  # Guarda ese valor
ifFalse x goto else_expr
# x = 1
<do then_expr>
```

```
f = <then_expr_result> # El resultado final de la expresion if
goto endif

# x = 0
label else_expr
<do_else_expr>
f = <else_expr_result> # El resultado final de la expresion if

label endif
```

Let In

Cool Input

```
let <id1>:<type1>, ... <idn>:<typen> in <expr>
```

CCIL Output

```
# Inicializa todas las variables let, tengan expresión o no
<init_let_var1>
<init_let_var2>
...
<init_let_varn>
# traduce la expresion en operacions
<do_in_expr>
f = <in_expr_fval> # Almacena el resultado final de la expresion let
```

Case Of

Cool Input

```
case <case_expr> of
    <id1>:<type1> => <expr1>
    <id2>:<type2> => <expr2>
    ...
    <idn>:<typen> => <exprn>
esac
```

CCIL Output

```
<init id1>
<init id2>
...
<init idn>

<do_case_expr>
x = <case_expr_result>
t = typeof x

# Analiznado rama 1
t1 = typeof <id1>
b1 = t1 == t # Comparando tipos
if b1 goto branch1: # En caso de exito ve a la rama

# Analizando rama 2
```

```
t2 = typeof <id2>
b2 = t2 == t
if b2 goto branch2

...

# Analizando rama n
tn = typeof <idn>
bn = tn == t
if bn goto brannch

<throw_runtime_exception> # Lanza una excepcion en ejcucion si no se ejecuta
ninguna rama


# Realizando logica the rama1
label branch1
<do_expr1>
goto end_case

# Realizando logica the rama2
label branch2
<do_expr2>
goto end_case

...

# Realizando logica the raman
label branchn
<do_exprn>
goto end_case

label end_case
```

Despacho Estático

Cool Input

```
<func_id>(<arg1>, <arg2>, ..., <argn>);
```

CCIL Output

```
<init arg1>
<init arg2>
...
<init argn>
r = call <func_id> n
```

Despacho Dinámico

Cool Input

```
<type1>@<type2>.<func_id>(<arg1>, <arg2>, ..., <argn>);
```

CCIL Output

```
<init arg1>
<init arg2>
...
<init argn>
t = allocate <type2> # It needs to give the same attributes that type one has
r = vcall t <func_id>  n
```

Declaración de un método

Cool Input

```
<function_id>(<arg1>:<type1>, ..., <argn>:<typen>) : <return_type>
{
    <function_expression>
}
```

CCIL Output

```
function <function_id> {
    param <arg1>
    param <arg2>
    ...
    param <argn>
    local <id1>
    local <id2>
    ...
    local <idn>
    <do_expresion>
    r = <expression_result>
    return r
}
```

Expresión de Bloque

Cool Input

```
{
    <expr1>;
    <expr2>;
    ...
    <exprn>;
}
```

CCIL Output

```
<init expr1 locals>
<init expr2 locals>
...
<init exprn locals>

<do_expr1>
<do_expr2>
...
<do_exprn>
```

Expresiones Aritméticas

Cool Input

```
3 + 5
```

CCIL Output

```
t = 3 + 5
```

More than one

Cool Input

```
3 + 5 + 7
```

CCIL Output

```
# Naive
t1 = 5 + 7
t2 = 3 + t1
```

Using non commutative operations

```
3 - 5 - 7
# -2 -7
# -9
```

```
t = 3 - 5
t = t - 7
```

Cool Input

```
100 / 20 / 5 / 2
```

CCIL Output

```
t = 100 / 20
t = t / 5
t = t / 2
```

Lenguaje CCIL

Definición del lenguaje CCIL. Tomamos como No Terminales sólo las palabras que empiecen con mayúsculas. El resto de palabras y símbolos se consideran como Terminales.

Program	→	.type TypeList .code CodeList
TypeList	→	Type | Type TypeList
Type	→	FeatureList
FeatureList	→	Attribute | Function | FeatureList | ϵ
	|	Attribute; FeatureList | Function; FeatureList
CodeList	→	FuncCode CodeList | ϵ
AttrCode	→	id {LocalList OperationList}
FuncCode	→	id {
		ParamList
		LocalList
		OperationList}
OperationList	→	Operation; OperationList | ϵ
Operation	→	id = ReturnOp
	|	goto id
	|	label id
	|	return Atom
	|	setattr id id Atom
	|	if Atom goto id
	|	ifFalse Atom goto id
	|	arg id
ReturnOp	→	Atom + Atom
	|	Atom - Atom
	|	Atom * Atom
	|	Atom / Atom
	|	not Atom
	|	neg Atom
	|	call id
	|	vcall typeId id
	|	typeof id
	|	getatrr id id
	|	allocate typeId
	|	Atom < Atom
	|	Atom <= Atom
	|	Atom = Atom
	|	allocate typeId
	|	getattr id id
	|	Atom
Atom	→	Constant | id
Constant	→	integer | string