Análisis semántico durante el proceso de compilación

Florencia Luciana Brunello, Andrés Valentino Ferrero, Julieta Paola Storino

Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba

Resumen En este informe presentaremos una descripción general de la etapa de análisis semántico en el proceso de compilación, basado en el libro *"Modern Compiler Implementation in ML"* [1]. Dicho texto expone técnicas, estructuras de datos y algoritmos fundamentales para la implementación de compiladores mediante el desarrollo progresivo de un compilador para *Tiger*, un lenguaje de programación didáctico de la familia de Algol creado por el autor. El compilador se implementa utilizando ML, un lenguaje funcional con tipado estático, estricto y modular que facilita la representación de las estructuras internas del compilador.

1. Introducción

Un compilador es una herramienta fundamental en el proceso de transformación del código fuente en ejecutable [2]. Este proceso se organiza en una secuencia de fases (*Figura 1*), que permiten descomponer y analizar progresivamente el programa. Antes de llegar a la etapa del análisis semántico, es necesario atravesar las siguientes fases previas:

1. **Lex**: divide el código fuente en unidades léxicas o *tokens*.
2. **Parse**: analiza la estructura sintáctica del programa.
3. **Parsing Actions**: construye los nodos del *Abstract Syntax Tree* (AST) correspondientes a cada frase analizada.

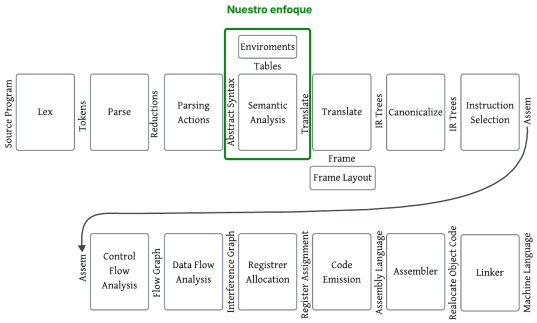

Figura 1. Fases de un compilador y sus interfaces.

2. Tablas de Símbolos

El análisis semántico de un compilador se caracteriza principalmente por el uso de *symbol tables* o también llamados *environments*. Un environment es un conjunto de bindings (asociaciones) denotadas por $\mapsto$, que vinculan identificadores de tipos, variables y funciones a sus respectivos tipos y ubicaciones. Por otro lado, cada variable tiene un *scope* (alcance) donde es visible. Cada variable local tiene un scope en el cual puede ser referenciada. Una vez finalizado el alcance de esta, la asociación a la variable es descartada del environment.

El manejo de environments puede implementarse con un estilo *imperativo* o *funcional*, independientemente del lenguaje que se esté compilando o del estilo de implementación del compilador.

2.1. Tablas Imperativas de Símbolos

En las tablas imperativas de símbolos se mantiene un único environment global y una *undo-stack* con suficiente información para deshacer las actualizaciones. Cuando se añade un símbolo al environment, actualizando este último, también se agrega a la pila. Al final de un scope, los símbolos extraídos tienen su última asociación eliminada, restaurando la asociación anterior.

Debido a que un programa puede tener muchos identificadores distintos, las tablas de símbolos deben permitir búsquedas eficientes. Es por esto que los environments suelen implementarse utilizando *tablas hash con encadenamiento externo*, donde las colisiones se resuelven mediante listas enlazadas en cada bucket.

De esta manera, la operación $\sigma_{i+1} = \sigma_i + a \mapsto \tau$ se implementa insertando τ en la tabla hash con la clave a, siempre que a no haya sido definida previamente. En caso contrario, se crea un *bucket* que apunta al valor anterior, de modo que, al buscar la variable, se obtenga τ como valor actual.

2.2. Tablas Funcionales de Símbolos

En las tablas funcionales de símbolos cada actualización produce una nueva versión del environment, manteniendo inmutable la anterior. Al finalizar un scope, simplemente se descarta la versión extendida, recuperando automáticamente el environment anterior sin necesidad de realizar operaciones de deshacer.

Las tablas hash, diseñadas para mutabilidad no son ideales en entornos funcionales, donde se requieren actualizaciones no destructivas. En cambio, se utilizan *árboles de búsqueda binaria* que permiten actualizaciones inmutables y eficientes. Además, cada nueva versión de la estructura comparte nodos con la versión anterior, evitando copias completas.

De esta manera, la operación $\sigma_{i+1} = \sigma_i + \{a \mapsto \tau\}$ no altera la tabla original, sino que construye una nueva versión que extiende σ_i con la asociación $a \mapsto \tau$. Internamente, esto se implementa compartiendo la estructura previa y enlazando el nuevo par (a, τ) al nodo raíz de la tabla. Así, al consultar la variable a en σ_{i+1}, se obtiene directamente τ, mientras que σ_i permanece intacta para usos posteriores o para restaurarse automáticamente al salir de un *scope*.

2.3. Tabla de Símbolos en el compilador Tiger

En el compilador de Tiger, los entornos se implementan con un estilo funcional. El módulo `SYMBOL` define una interfaz polimórfica para manejar entornos. Por cuestiones de eficiencia en las búsquedas, cada `string` es convertido a un `symbol`. Las tablas (`'a table`) permiten asociar símbolos con diferentes tipos de información (valores, tipos, etc.), donde `enter` añade asociaciones (creando una nueva tabla) y `look` permite buscar `el binding` de un símbolo.

```
signature SYMBOL =
sig
   eqtype symbol
   val symbol : string -> symbol
   val name : symbol -> string

   type 'a table
   val empty : 'a table
   val enter : 'a table * symbol * 'a -> 'a table
   val look  : 'a table * symbol -> 'a option
end
```

Programa 1.1. Firma del módulo SYMBOL

Internamente, cada símbolo se representa como un par `(string` $\times$ `int)`. Un contador llamado `nextsym`, junto con una tabla hash, se encarga de que cada entero sea único a cada cadena. Esta representación facilita comparaciones más eficientes y permite utilizar `IntBinaryMap` para implementar las tablas.

```
structure Symbol :> SYMBOL =
struct
  type symbol = string * int

  exception Symbol
  val nextsym = ref 0
  val hashtable : (string,int) HashTable.hash_table =
    HashTable.mkTable(HashString.hashString, op =)
                      (128, Symbol)

  fun symbol name =
    case HashTable.find hashtable name
      of SOME i => (name, i)
       | NONE => let val i = !nextsym
                 in nextsym := i+1;
                   HashTable.insert hashtable (name, i);
                   (name, i)
                 end

  fun name(s, n) = s

  type 'a table = 'a IntBinaryMap.map
  val empty = IntBinaryMap.empty
```

```
   fun enter(t: 'a table, (s, n): symbol, a: 'a) =
        IntBinaryMap.insert(t, n, a)
   fun look(t: 'a table, (s, n): symbol) =
        IntBinaryMap.look(t, n)
end
```

Programa 1.2. Implementación de SYMBOL

3. Asociaciones para el compilador Tiger

En el lenguaje Tiger existen dos *namespaces*: uno para los tipos y otro para las variables y funciones. Un identificador de tipo se asocia a un elemento del tipo `Types.ty`, definido en el módulo `Types`, el cual representa la estructura de tipos del lenguaje.

```
    structure Types =
    struct
        type unique = unit ref
        datatype ty = INT
                    | STRING
                    | RECORD of (Symbol.symbol * ty) list *
                      unique
                    | ARRAY of ty * unique
                    | NIL
                    | UNIT
                    | NAME of Symbol.symbol * ty option ref
    end
```

Programa 1.3. Definición del módulo Types

También define dos clases de tipos: los primitivos (`int` y `string`) y los tipos construidos, que pueden ser `record` o `array` de otro tipo. Los records están compuestos por campos con nombre y tipo. Como cada expresión de tipo record representa una construcción distinta, Tiger los distingue mediante un valor especial llamado `unique`.

3.1. Environments

El tipo de tabla definido en el módulo `Symbol` permite mapear símbolos a sus respectivas asociaciones. A partir de esto, se definen dos entornos: uno de tipos y otro de valores. Un entorno de tipos es una estructura que mapea símbolos a elementos de tipo `Types.ty`, es decir, una `Types.ty Symbol.table`.

En el caso de los identificadores de valor, es necesario distinguir entre variables y funciones. Una variable estará asociada a su tipo, mientras que una función estará asociada a los tipos de sus parámetros y a su tipo de retorno. Esta información se encapsula en el tipo `enventry`, y un entorno de valores es una tabla que mapea símbolos a entradas de tipo `enventry`.

El módulo `Env` define dos entornos base: `base_tenv`, que representa el entorno de tipos predefinido (por ejemplo, mapeando `int` a `Ty.INT` y `string` a

`Ty.STRING`); y `base_venv`, que contiene el entorno de valores predefinidos, como las funciones incorporadas.

```
signature ENV =
sig
    type access
    type ty
    datatype enventry = VarEntry of {ty:ty}
                      | FunEntry of {formals: ty list,
                          result:ty}
    val base_tenv : ty Symbol.table
    val base_tenv : enventry Symbol.table
end
```

Programa 1.4. Firma del módulo ENV

Una variable se asocia a una entrada del tipo `VarEntry`, que contiene la información sobre su tipo. En el caso de una función, se utiliza una `FunEntry`, que almacena dos elementos clave: `formals`, que representa la lista de tipos de los parámetros, y `result`, que indica el tipo de retorno de la función (o `UNIT` si no devuelve un valor). Para realizar la verificación de tipos, únicamente se necesita conocer el `formals` y el `result` de cada función.

4. Verificación de tipos de expresiones

Durante la etapa de *parsing actions*, se construye el *Abstract Sintax Tree*, donde se valida la sintaxis del programa. Este árbol se utiliza como punto de partida para realizar la verificación semántica, es decir, comprobar la corrección del programa desde el punto de vista del uso de los identificadores y tipos.

La verificación incluye aspectos como: el uso de variables o funciones no declaradas, que cada uso esté dentro del alcance correcto, que las llamadas a funciones tengan la aridad adecuada y tipos de argumentos compatibles, y que no haya declaraciones duplicadas, entre otros.

A partir de esta etapa se define una nueva estructura, el módulo `Semant`, que contiene cuatro funciones encargadas de recorrer el árbol sintáctico y realizar estos chequeos.

```
type venv = Env.enventry Symbol.table
type tenv = Types.ty Symbol.table

val transVar : venv * tenv * Absyn.var -> expty
val transExp : venv * tenv * Absyn.exp -> expty
val transDec : venv * tenv * Absyn.dec -> {venv: venv, tenv:
    tenv}
val transTy  :        tenv * Absyn.ty  -> Types.ty

type expty = {exp: Translate.exp, ty: Types.ty}
(*Translate.exp es la traduccion de una exp a codigo
    intermedio, junto a su tipo Tiger asociado*)
```

```
structure Translate = struct type exp = unit end
(*dummy Translate. Usaremos () para cada exp ya que aqui no
    nos interesa esa parte*)
```

Programa 1.5. Corrección de tipos dentro del módulo Semant

El chequeo de tipos se realiza mediante una función recursiva que recorre el árbol de sintaxis abstracta. Esta función se denomina `transExp`, ya que, en etapas posteriores de la compilación, también se encargará de traducir las expresiones a *código intermedio*.

Durante este proceso, se utilizan dos entornos: `type-env`, que permite obtener el tipo real de un identificador de tipo definido en `Types.ty` (usado principalmente en declaraciones de tipos), y `value-env`, que se emplea para verificar los tipos de variables y funciones conforme aparecen en el código fuente.

A continuación, se muestra el caso de la suma $e_1 + e_2$ como ejemplo de este análisis:

```
fun transExp(venv, tenv,
             Absyn.OpExp{left, oper=Absyn.PlusOp,
                 right,pos}) =
    let val {exp=_, ty=tyleft} = transExp(venv,tenv,left)
        val {exp=_, ty=tyright} = transExp(venv, tenv,right)
     in case tyleft of Types.INT => ()
              |_ => error pos "integer␣required";
        case tyright of Types.INT => ()
              |_ => error pos "integer␣required";
        {exp=(), ty=Types.INT}
    end
```

Programa 1.6. Implementación parcial de la función transExp para la operación suma

Se aplica recursión sobre cada subexpresión, y se informa un error de tipos si alguno de los operandos de la suma no es de tipo `int`. Este chequeo se realiza utilizando los entornos correspondientes y comparando los tipos inferidos de las expresiones con el tipo esperado.

Si ambos operandos son de tipo `int`, la función devuelve la expresión tipada `expty (), Types.INT`, indicando que el resultado es una expresión entera. Este procedimiento se aplica de forma análoga para otras operaciones aritméticas.

A continuación, veremos cómo se realiza el chequeo de tipos para variables. Para esto, es importante tener en cuenta que `transExp` recorre expresiones del tipo `Absyn.exp`, mientras que `transVar` recorre variables del tipo `Absyn.var`. Dado que ambas funciones son recursivas, se definen utilizando la construcción `and`.

```
structure A = Absyn
...
fun transExp(venv, tenv) =
    let fun trexp (A.OpExp{left,oper=A.PlusOp,right,pos}) =
                ...
          | trexp (A.RecordExp ...) ...
        and trvar (A.SimpleVar (id,pos)) =
```

```
        (case Symbol.look(venv,id)
          of SOME(E.VarEntry{ty}) =>
              {exp=(), ty=actual_ty ty}
          | NONE => (error pos ("undefined␣variable"
                                ^ S.name id);
      | trvar (A.FieldVar(v,id,pos)) = ...
in trexp
```

Programa 1.7. Versión limpia y parcial de la función transExp

La cláusula de `trvar` que verifica el tipo de una `SimpleVar` muestra cómo se utilizan los entornos para determinar el alcance de una variable. Si el identificador se encuentra en el entorno y está asociado a una `VarEntry` (y no a una `FunEntry`), entonces su tipo es el especificado en dicha `VarEntry`.

Puede ocurrir que el tipo almacenado en la `VarEntry` sea un tipo `NAME`. Sin embargo, todos los tipos que retorna `transExp` deberían ser tipos reales, es decir, tipos con sus nombres ya resueltos hasta llegar a sus definiciones subyacentes. Por esta razón, resulta útil contar con una función denominada `actual_ty`, que se encarga de seguir la cadena de referencias `NAME` hasta obtener un tipo concreto. El resultado será un valor de tipo `Types.ty` que no sea un `NAME`.

5. Verificación de tipos en declaraciones

Los entornos son construídos y aumentados por declaraciones. En Tiger, sólo existen declaraciones del tipo `let`. El chequeo de tipos se hace de la siguiente manera:

```
      | trexp(A.LetExp{decs, body,pos}) =
          let val {venv=venv', tenv=tenv'} =
                    transDec(venv, tenv, decs)
          in transExp(venv', tenv') body
        end
```

Programa 1.8. Caso LetExp en la función trexp que actualiza entornos y evalúa el cuerpo

En este contexto, `transExp` extiende los entornos `venv` y `tenv`, generando nuevas versiones `venv'` y `tenv'` que se utilizan para verificar los tipos dentro del cuerpo de la expresión (`body`). Una vez finalizado este análisis, los entornos extendidos se descartan, ya que su alcance está limitado al bloque correspondiente.

La función clave en este proceso es `transDec`, que devuelve un par de entornos extendidos con los nuevos *bindings*, los cuales son válidos únicamente dentro del ámbito de la declaración.

Dentro de una expresión `let`, pueden aparecer tres tipos de declaraciones: de variables, de tipos y de funciones. En el caso de las declaraciones de variables, el lenguaje Tiger permite especificar el tipo explícitamente o dejar que se infiera a partir de la expresión de inicialización. A continuación, analizaremos este último caso:

```
fun transDec (venv,tenv,A.VarDec{name, typ=NONE,
    init,...}) =
    let val {exp,ty} = transExp(venv,tenv,init)
     in {tenv=tenv, venv=S.enter
         (venv,name,E.VarEntry{ty=ty})}
    end
```

Programa 1.9. Caso `VarDec` sin tipo explícito en la función `transDec`

En el caso de las declaraciones `let`, se obtiene el tipo de la expresión de inicialización para determinar el tipo de la variable. Para ello, se pasa la expresión `init` como parámetro a la función `transExp`. Luego, simplemente se agrega la nueva variable al entorno de valores como una entrada del tipo `VarEntry`.

Para las declaraciones de tipo, el procedimiento es distinto y se detalla a continuación:

```
| transDec (venv,tenv,A.TypeDec([{name,ty}]) =
      {venv=venv,
      tenv=S.enter (tenv,name, transTy(tenv, ty))}
```

Programa 1.10. Chequeo de tipos

La función `transTy` traduce las expresiones de tipo tal como aparecen en la sintaxis abstracta (`Absyn.ty`) a las descripciones de tipo 'reales' (`Types.ty`). Es decir, añadimos al entorno de tipos, el tipo declarado con su tipo respectivo en `Types.ty`.

Por último, consideremos el caso de la verificación de tipos en las declaraciones de funciones:

```
| transDec(venv, tenv,
          A.FunctionDec[{name, params, body, pos,
              result=SOME(rt,pos)}]) =
      let val SOME(result_ty) = S.look(tenv,rt)
          fun transparam{name, typ,pos} =
                case S.look(tenv,typ)
                of SOME t => {name=name, ty=t}
          val params' = map transparam params
          val venv' = S.enter (venv,name,
                      E.FunEntry{formals= map #ty
                          params', result=result_ty})
          fun enterparam ({name,ty},venv) =
                      S.enter (venv, name,
                          E.VarEntry{access=(), ty=ty})
          val venv'' = fold enterparam params' venv'
      in transExp(venv'', tenv) body;
      end
```

Programa 1.11. Caso `TypeDec` en la función `transDec` que actualiza el entorno de tipos

Consideremos la siguiente declaración de función en Tiger:

```
function f(a: ta, b: tb) : rt = body
```

El procesamiento de esta declaración en `transDec` comienza buscando en el entorno de tipos el tipo real de `rt`, que representa el tipo de retorno de la función. Luego, se construye una nueva lista de parámetros `params'`, donde a cada parámetro formal se le asocia su tipo real, también obtenido a partir del entorno de tipos.

De esta forma, se genera una lista de pares del tipo `(a, ta)`, `(b, tb)`, donde cada identificador de parámetro está asociado a su tipo correspondiente. Si alguno de los tipos no está definido en `tenv`, se lanza una excepción.

Con esta información, se actualiza el entorno de valores (`venv`) agregando una nueva entrada del tipo `FunEntry`, que contiene la lista de tipos de los parámetros y el tipo de retorno. Así se obtiene un nuevo entorno `venv'`.

Para poder verificar el cuerpo de la función, es necesario extender aún más `venv'` agregando cada parámetro como una `VarEntry`. Esto se logra mediante `fold`, utilizando una función auxiliar `enterparam` que inserta cada variable en el entorno de valores.

Una vez construido este entorno extendido (`venv''`), se llama recursivamente a `transExp` para analizar el cuerpo de la función. Al finalizar esta verificación, `venv''` se descarta y `transDec` retorna el par `venv', tenv`. Este nuevo entorno podrá utilizarse luego para analizar expresiones que hagan uso de la función `f`.

Referencias

1. Appel, A. W. (2004). Modern compiler implementation in ML (Rev. y ed. expandida). Cambridge University Press.
2. Thain, D. (2023). Introduction to compilers and language design (2a ed.). http://compilerbook.org
3. Appel, A. W. (1998). Modern compiler implementation in Java. Cambridge University Press.
4. Harper, R. (1986). Introduction to Standard ML. Carnegie Mellon University.