Chequeos Estáticos

Matias Gobbi

17 de enero de 2020

Fundamentos

Una vez implementada la sintaxis del lenguaje y el parser para el intérprete, es hora de comenzar la etapa del análisis semántico. Describiremos los distintos chequeos estáticos que el intérprete deberá realizar para poder determinar que un fragmento de código es un programa válido en el lenguaje. También haremos mención de algunos chequeos dinámicos que puede ser conveniente implementar.

1. Introducción

El objetivo de este informe es servir como documentación en el desarrollo del intérprete para $\Delta\Delta$Lang. En el mismo, describiremos los distintos aspectos a implementar en el análisis semántico, principalmente los chequeos estáticos y algunos dinámicos. La idea es que el intérprete sea más robusto, y pueda compilar (y ejecutar) solo programas válidos del lenguaje.

Mientras que el parser se encarga de filtrar aquellos programas que no estén bien formados sintácticamente, el trabajo de este nuevo análisis es rechazar aquellos fragmentos de código que presenten errores semánticos. Los chequeos estáticos comprobarán aspectos como la adecuación de los tipos de datos empleados, mientras que los dinámicos, el uso correcto de memoria.

Para dar un formato estructurado al trabajo, el mismo se dividirá en cuatro secciones correspondientes a cada una de las principales construcciones sintácticas del lenguaje. Probablemente, a la hora de la implementación, esta división no será respetada. También se hará mención de distintos aspectos que no fueron definidos todavía, los mismos están pendientes a ser debatidos en el transcurso del desarrollo del intérprete.

2. Expresiones

A continuación mencionamos los distintos chequeos que se deberán realizar para las expresiones del lenguaje. Las mismas se dividen en el chequeo de tipos, y en los chequeos dinámicos. También se detallan algunas cuestiones pendientes a resolver.

2.1. Chequeos de Tipos

Para el chequeo de tipos deberíamos analizar aspectos como el uso de operadores de forma adecuada, la coersión de tipos, o la especificación de argumentos de forma apropiada. A continuación se enumeran los mismos.

En la llamada a una función, la misma debe ser invocada con la cantidad correcta de parámetros. En el lenguaje no se utilizan funciones parciales por lo que llamar una función con parámetros de menos no se debería permitir. Al mismo tiempo, la llamada con entradas de más carece de sentido, por lo que no debe ser aceptada.

```
var x: int
x := factorial(10, 5)
```

Con el acceso a arreglos tenemos una tarea similar a la de funciones. Solo deberíamos permitir el acceso cuando se especifican la misma cantidad de índices que dimensiones posee el mismo. Si se pasan índices de más, no tendría sentido el acceso, y si se pasan de menos, no se trabaja con arreglos parciales en el lenguaje.

```
var a: array [1..5] of int
var x: int
x := a[1, 5]
```

Los tipos de los operandos deben coincidir con los esperados por el operador. En particular, en el lenguaje hay definidos operadores aritméticos, los cuales trabajan solo con tipos enteros y reales, y operadores booleanos que justamente trabajan con valores booleanos. Intentar sumar un entero con un caracter debería resultar en un error de compilación, por ejemplo.

```
var x: int
x := 4 + 'a'
```

Otro problema con operadores reside en los de igualdad y orden. No queda claro aún para que clase de valores deberían estar definidos. En primera instancia, los tipos básicos y los constantes deberían ser igualables y ordenables. De todas formas, esta decisión queda pendiente para el desarrollo futuro del intérprete.

```
var b: bool
b := Lunes == Martes
```

Tampoco queda claro, si el lenguaje debería implementar subtipado de entero a reales. Ya que hay operadores numéricos que trabajan con ambos tipos, sería deseable permitir emplear ambos en ciertas ocasiones. Una alternativa, podría ser aplicar conversiones de tipo de forma implícita cuando sea necesario. Una última opción, es tratar a los enteros y a los reales como dos conjuntos completamente distintos, y no permitir ninguna operación entre ambos.

```
var x: real
x := 4 + 6.0
```

Los tipos de las expresiones deben ser adecuados al contexto en el que son evaluadas. En particular, para los argumentos de funciones e índices de arreglos deben coincidir al esperado. Los tipos de los parámetros de una función son especificados en su prototipo. Mientras, el tipo de los índices de un arreglo es inferido en el momento de su declaración. Hay tres posibilidades para los mismos, índices de enteros, de constantes o de caracteres.

```
var x: int
var y: array [1..5] of int
x := factorial(true)
x := y[true]
```

Hay cuatros clases de locations distintas, y cada una posee operadores que trabajan con las mismas. Debería haber un chequeo para asegurar que los operadores aplicados a los distintos identificadores coincida con el declarado en el programa. Ya sea para punteros, tuplas o arreglos.

```
var a: array [1..5] of int
var x: int
x := a.field + #x
```

2.2. Chequeos Dinámicos

Para los chequeo dinámicos deberíamos analizar situaciones donde se accede a memoria no válida o donde no hay un valor definido. También se debe trabajar con precaución cuando se usan operadores que no están completamente definidos.

En el acceso a arreglos, los índices empleados deben ser válidos. Es decir, no podemos acceder fuera del rango de un arreglo. Este análisis debe ser dinámico ya que primero se debe evaluar una expresión antes de saber a que índice hace referencia.

```
var a: array [1..5] of int
var x: int
x := a[1 + factorial(10) * 3]
```

Otro chequeo dinámico es el de comprobar la inicialización de una location antes de su uso. Para las cuatro clases de locations, se debería validar que efectivamente almacenan un valor en el lugar que se intenta acceder antes de su lectura.

```
var a: array[1..5] of int
var x: int
x := a[1]
```

Una última evaluación, es el chequeo de divisiones. Antes de aplicar la operación de división, o módulo, se debe revisar que el divisor no sea cero. De lo contrario, tendríamos una operación que no se puede evaluar.

```
var x: int
x := 10 / (5 - 5)
```

3. Sentencias

Para las sentencias del lenguaje, los chequeos se basan principalmente en las instrucciones for y los procedimientos. Al utilizar expresiones, también es necesario realizar chequeos de tipos en estas construcciones.

3.1. Procedimientos

Las validaciones que se deben realizar en los procedimientos son similares a las hechas para las funciones en la sección anterior. A continuación se listan las mismas.

En la llamada a un procedimiento, el mismo debe ser invocado con la cantidad correcta de parámetros. De lo contrario, no se puede ejecutar la instrucción. Independientemente si la cantidad de entradas es menor o mayor a la esperada, en ambos casos la evaluación carecería de sentido.

```
var a: array [1..5] of int
inicializarArreglo(a)
swap(a, 1)
```

Además, al requerir valores de entrada de cierto tipo, es necesario realizar un chequeo de tipos para asegurar la coincidencia de los mismos. Esta validación es similar a la hecha para las funciones.

```
var b: bool
inicializarArreglo(b)
```

Las instrucciones **alloc** y **free** son una clase especial de procedimientos. Sintácticamente, el parser limita la especificación de los mismos para que solo tomen una location como parámetro de entrada. La tarea del análisis semántico es entonces, asegurar que la location sea efectivamente un puntero y no otra clase de variable. Es decir, necesitamos otro chequeo de tipos.

```
var b: bool
alloc(b)
```

3.2. Chequeos de For

Hay distintas versiones para la instrucción *for*. Cada una requiere de un conjunto distinto de validaciones para asegurar su correcta especificación. A continuación se detallan las mismas.

La sentencia *for* posee una variable denominada *iteradora*, que es declarada, inicializada y modificada a lo largo de la ejecución de la instrucción. La misma debería utilizar un identificar nuevo, que no se encuentre en el scope actual del programa en ejecución.

```
var x: int
for x := 1 to 5 do
  skip
do
```

La variable iteradora declarada en el encabezado de una instrucción *for* no debería ser modificada en el cuerpo del mismo. Hacer esto es considerado una mala práctica ya que dificulta la comprensión del algoritmo. Esta validación podría informar con un *warning* estas situaciones.

```
for x := 1 to 5 do
  x := 8
od
```

En el caso de la instrucción *for* para elementos iterables queda pendiente definir que se considera *iterable* en el lenguaje. Esto fue discutido en grupo, pero aún no se ha llegado a un concenso, por lo que deberá ser determinado a lo largo del desarrollo del intérprete. Además de definir que se considera iterable, queda implementar el chequeo correspondiente para determinar que algo es iterable.

```
var a: array [1..5] of int
inicializarArreglo(a)
for x in a do
  skip
od
```

Para las instrucciones *for* que operan en rangos, es necesario un chequeo de tipos. Se debe evaluar si los tipos de ambos límites coinciden. No tendría sentido comenzar en un valor numérico y terminar con un valor constante, por ejemplo.

```
for x := 1 to Martes do
  skip
od
```

Finalmente, y agregando a lo anterior, también se debe verificar que el tipo de los límites sea enumerable. Mientras que los números enteros tienen definidos el sucesor y el predecesor, los números reales no lo poseen y por lo tanto, no pueden ser utilizados como límites.

```
for x := 1.0 to 5.5 do
  skip
od
```

3.3. Otras Instrucciones

Otra clase de validaciones corresponden al chequeo de tipos para sentencias conformadas por expresiones. Ejemplos de estas serían las asignaciones, los condicionales *if* y los iteradores *while*.

En el caso de la asignación, se debería comprobar que los tipos de la location a asignar y la expresión a evaluar coincidan. Ambos chequeos, por separado, son

realizados en la sección de expresiones. Pero se debe realizar una verificación extra para asegurar que la instrucción es válida.

```
var x: int
x := true
```

Para las instrucciones *if* y *while*, se debe verificar que la expresión condicional tenga tipo booleano. De lo contrario, no tendría sentido ejecutar la instrucción correspondiente.

```
var a: array [1..5] of int
inicializarArreglo(a)
if 4 * 6 then
  selectionSort(a)
fi
```

3.4. Chequeos Dinámicos

Para esta sección, a diferencia de la descripta para expresiones, los chequeos dinámicos serían opcionales. El objetivo no es asegurar el correcto funcionamiento del intérprete, ya que estas validaciones no lo garantizan. Sino, más bien, poder agregar funcionalidades extras y mensajes de advertencia.

Un chequeo opcional que se podría realizar sobre todas las clases de procedimientos, es revisar que al invocar uno, todas las entradas que tengan la etiqueta **in** estén debidamente inicializadas. Esto sería opcional ya que hay un chequeo en las expresiones que se encarga de esta tarea. La idea es poder informar al usuario de un posible error antes de tener que comenzar a ejecutar el método.

```
var a: array [1..5] of int
swap(a, 1, 5)
```

La otra verificación que se podría agregar es el seguimiento del uso de memoria del programa. Al emplear punteros, el programa irá reservando y liberando memoria en la medida que avanza su ejecución. Lo que podría ser beneficioso para el usuario, es poder ir observando de forma gráfica como se modifica la memoria utilizada. También sería una buena forma de evitar memory leaks en la ejecución de programas.

```
var p: pointer of int
alloc(p)
#p := 5
```

4. Tipos

Posiblemente el análisis más complejo es el de tipos. Para el mismo, se deben comprobar varios aspectos distintos para asegurar la correcta especificación del programa. En un futuro, es probable que la sintaxis para tipos del lenguaje sea reestructurada para poder facilitar la validación del código y evitar malas prácticas a la hora de la implementación de algoritmos.

4.1. Arreglos

Cuando se especifica un arreglo, se debe comprobar que los límites del mismo no definan un rango vacío. En el lenguaje no se hace ningún uso de tipos vacíos, por lo que un arreglo que no puede almacenar valores carece de sentido. Esta validación solo se debería hacer cuando se especifican límites fijos, como enteros, constantes, o caracteres.

```
var a: array [1..0]
```

Otra validación necesaria para límites de arreglos es el chequeo de tipos. Ambos límites deben poseer el mismo tipo, de lo contrario no se podrían enumerar los índices válidos. En el caso de usar límites variables, el tipo de los mismos es inferido en base al contexto del programa.

```
var a: array [1..Martes]
```

El siguiente chequeo, es tanto para los tipos como para los programas. La especificación de límites variables de arreglos se debería permitir solo en el prototipo de métodos, ya que sería el único lugar donde se podría inferir el verdadero valor de los mismos. Una vez especificado, un límite variable puede seguir siendo utilizado en el cuerpo de una función como una variable más, la cual no debe ser modificada; o en el cuerpo de un procedimiento como una entrada de solo lectura, que no podrá ser alterada. Obviamente, al poder utilizarse como una variable más, su nombre debería ser único en el scope de ejecución del programa.

```
fun factorial (n: int) ret fact: int
  var a: array [i..j] of int
  ...
end fun
```

Un último comentario sobre arreglos, es un detalle que queda pendiente a discutir ¿Deberían los límites ser más expresivos? Es decir, podríamos querer definir el rango de un arreglo en base al valor de una variable definida en el programa. Por como está pensado el lenguaje, en una primera instancia no se puede hacer esto. Si se permitiera, los chequeos anteriores deberían ser modificados para adecuarse al nuevo funcionamiento buscado. Esta idea, por ahora, es un simple comentario sobre el diseño actual del ΔΔLang.

```
fun factorial (n: int) ret fact: array [0..n] of int
  fact[0] := 1
  for i := 1 to n do
    fact[i] := i * fact[i-1]
  od
end fun
```

4.2. Tipos en General

Ciertos valores deben ser definidos antes de su uso. Cuando se accede al campo de una tupla, el alias utilizado debe estar definido. De lo contrario, la operación no tendría sentido. Sucede algo similar cuando se emplean valores constantes. Antes tiene que haber sido definido un tipo que posea el valor utilizado.

```
var x: int
var p: persona
x := p.numFavorito
```

Los identificadores utilizados en el programa deben ser únicos en el scope de ejecución. De lo contrario, no se puede saber a que clase de construcción se está haciendo referencia. Hay una salvedad cuando el contexto provee la suficiente información como para desambigüar la situación. De todas formas, los nombres de nuevos tipos de datos deben ser distintos entre sí. Además, los nombres de constantes no pueden ser repetidos en un mismo programa, de lo contrario, no se puede saber a que tipo de valor se refieren.

```
type list = array [1..10] of int
type list = tuple
            value: int
            next: pointer of list
            end tuple
```

Otra verificación consiste en comprobar que al emplear un tipo paramétrico anteriormente definido, el mismo sea correctamente instanciado. Si se pasan parámetros de menos, no se puede evaluar el mismo. Si se pasan de más, no se pueden emplear la totalidad de estos.

```
var x: count of (int, real)
```

Un último análisis consiste en el uso de variables de tipo. Este chequeo sería adecuado también en la sección de programas. Se debe determinar en que situaciones se permite el uso de variables de tipo, y que se puede deducir o inferir de las mismas. En particular, uno podría usar variables de tipo para un algoritmo de ordenación pero estas no podrían tomar cualquier valor ya que necesitan emplear las operaciones de ordenación e igualdad para el correcto funcionamiento del programa.

```
fun indiceMinimo (a: array[1..n] of T) ret min: int
  min := 1
  for i := 1 to n do
    if a[i] < a[min] then
      min := i
    fi
  od
end fun
```

4.3. Creación de Tipos

Las siguientes validaciones se podrían omitir si se reestructurara la sintaxis del lenguaje de tal forma para evitar esta situaciones. De todas formas, para no complicar de más la misma, el análisis semántico se puede encargar de controlar estos casos.

No se debería permitir definir un tipo estructurado en términos de otro de la misma clase. Los tipos estructurados del lenguaje son los punteros y los arreglos. No tendría mucho sentido definir un arreglo de arreglos, cuando en el mismo lenguaje ya se ofrecen construcciones para crear arreglos multidimensionales, por ejemplo. Esta precaución también se debería tener al trabajar con sinónimos de tipos.

```
var a: array [1..5] of array [6..10] of int
```

Al definir tipos paramétricos, se debería controlar que todas las variables de tipos especificadas sean efectivamente empleadas en el cuerpo de la definición. De lo contrario, tendríamos parámetros de más que no tienen uso en el nuevo tipo.

```
type count of (A) = int
```

Complementando con la verificación anterior, tampoco debería ser posible crear un nuevo tipo que posea variables de tipo libres en su definición. Si fuera posible esto, no se podría inferir su tipo ni tampoco crear valores para el mismo.

```
type matriz = array [1..5, 1..5] of A
```

Otra característica a ser debatida es la recursión en la definición de tipos. Todavía resta decidir hasta que punto se permite la misma en la creación de nuevos tipos de datos. Una primera opción es permitirla solo dentro de tuplas, cuando el campo es un puntero al mismo tipo. De esta forma, se podrían crear listas abstractas con los nuevos tipos.

```
type listaInfinita = tuple
                     valor: int,
                     sig: listaInfinita
                     end tuple
```

Otra restricción que se debería agregar es evitar la creación de sinónimos de tipos que sean simplemente un renombrado de una variable de tipo. Esta validación puede no ser indispensable para el correcto funcionamiento del intérprete, pero favorece la creación de código prolijo.

```
type sinonimo of (A) = A
```

Finalmente, otra verificación con un propósito similar al chequeo anterior. No se debería permitir la creación de tuplas con un solo campo, ya que definir un sinónimo de tipo consigue el mismo resultado y de forma más prolija. Por lo que la definición de tuplas debería ser permitida solo cuando se especifican dos o más campos.

```
type count = tuple
             field: int
             end tuple
```

5. Programas

Los chequeos para programas son más simples y fáciles de verificar en comparación a los de la sección anterior. Se pueden dividir en dos clases, validaciones en declaraciones y validaciones de métodos. A continuación se listan las mismas.

5.1. Chequeos en Declaraciones

Antes de usar una variable, la misma debe haber sido declarado junto a su tipo. De esta forma, el intérprete puede analizar el programa antes de ejecutar el mismo. Por lo tanto, si en un fragmento de código se utiliza un identificador nuevo sin previa definición, no se puede compilar el mismo. Hay que tener en cuenta que el prototipo de un método, al igual que una sentencia *for*, pueden declarar variables también.

```
var x: int
y := 8
```

Similar a lo anterior, las variables no son las únicas construcciones que deben ser declaradas antes de su uso. En particular, las distintas funciones, procedimientos, y tipos de datos que se utilizan en un programa también deben ser definidos previamente antes de su utilización.

```
var x: int
x := fibonacci(10)
```

Todas las construcciones anteriores deben ser definidas antes de su uso en el programa. Incluso si se garantizara esta condición, un programa podría seguir teniendo fallas que imposibilitarían su ejecución. En particular, los identificadores empleados para definir funciones, procedimientos y variables deben ser únicos entre los nombres utilizados para identificar elementos de su misma categoría. De lo contrario, si se utilizara la misma referencia para elementos distintos, el intérprete no podría adivinar a cual se está invocando.

```
var x: int
var x: real
x := 4 + 6.0
```

Una última validación sobre las declaraciones tiene que ver con los argumentos de los métodos. Los parámetros de entrada especificados en el prototipo de un procedimiento o función, y el retorno de este último, deben ser analizados como variables. Por lo tanto, sus identificadores también deben ser únicos en el scope de ejecución del programa.

```
fun factorial (n: int) ret fact: int
  var n: bool
  ...
end fun
```

5.2. Chequeos en Métodos

Un procedimiento recibe una serie de parámetros, algunos de lectura y otros de escritura, y realiza una serie de pasos en base a los valores de entrada, modificando los valores de salida. En definitiva, sería apropiado verificar que los argumentos que se especifican con la etiqueta *in* sean efectivamente empleados para lectura, mientras que los que posean la etiqueta *out* deberán ser modificados. Los argumentos que posean ambas, deberán ser sometidos a las dos clases de operaciones.

```
proc swap (in/out a: array[n..m] of T, in i, j: int)
  i := i + 1
  ...
end proc
```

Siguiendo con la verificación de procedimientos, todo procedimiento debería tener por lo menos un argumento de escritura. Ya que no producen valores de salida, sino que simplemente modifican el estado de las variables seleccionadas con la etiqueta *out*, sería carente de sentido permitir la especificación de procedimientos que no produzcan efectos secundarios.

```
proc suma (in i, j: int)
  var x: int
  x := i + j
end proc
```

En base a lo mencionado anteriormente, podemos hacer un paralelo con funciones. Las mismas, reciben una serie de argumentos y producen un nuevo valor que será devuelto al lugar donde se invocó. Para asegurar un correcto funcionamiento de un programa, se debería verificar que la variable de retorno sea efectivamente asignada en el cuerpo de la función.

```
fun factorial (n: int) ret fact: int
  var x: int
  if n > 1 then
    x := n * factorial(n - 1)
  fi
end fun
```

Otro paralelo que se puede establecer entre la validación de procedimientos y funciones, es comprobar que los parámetros de esta última no sean modificados dentro del cuerpo de la función. La idea, es que el comportamiento de una función solo dependa de los valores de sus argumentos y no modifique el estado

de las variables ajenas a la misma, por lo tanto, no debería ser posible alterar los parámetros en su cuerpo.

```
fun factorial (n: int) ret fact: int
  if n > 1 then
    n := n * factorial(n - 1)
  else
    n := 1
  fi
  fact := n
end fun
```

Para finalizar, los últimos chequeos tienen que ver con dos construcciones que solo pueden ser introducidas en el prototipo de métodos. Este aspecto ya se describió brevemente en la sección de tipos. Tanto las variables de tipo, como los límites variables para arreglos solo deberían ser especificados al comienzo de una función o procedimiento ya que sería el único lugar donde se podría inferir el valor de los mismos en tiempo de ejecución. Una vez definidos, se pueden emplear en el resto del cuerpo del método.

```
proc swap (in/out a: array[n..m] of T, in i, j: int)
  var tmp: T
  tmp := a[i]
  a[i] := a[j]
  a[j] := tmp
end proc
```