Facultad de Matemática, Astronomía, Física y Computación

Universidad Nacional de Córdoba

Semántica estática para un lenguaje *Pascal-like*

Matías Federico Gobbi

Director: Alejandro Emilio Gadea

Córdoba, Argentina 2021

Resumen

Este trabajo consiste en el diseño e implementación de un lenguaje de programación estructurado basado en el lenguaje *Pascal*, orientado al aprendizaje de algoritmos y estructura de datos. El mismo es utilizado actualmente en una materia de la *FaMAF*, contando con una definición informal. Existe una sintaxis concreta relativamente consolidada aunque no especificada, y la semántica está definida de manera intuitiva. En el trabajo se estudió la información disponible a partir del dictado de la materia obteniendo una definición formal de la sintaxis abstracta, en conjunto con la definición de varios chequeos estáticos, como el sistema de tipos.

Índice general

Capítulo 1

Introducción

En el siguiente proyecto desarrollaremos la semántica estática de un lenguaje de programación inspirado en el pseudocódigo utilizado por la materia *Algoritmos y Estructura de Datos II*. A diferencia de la mayoría de los lenguajes de programación conocidos, en el trabajo se presenta un desarrollo formal del lenguaje previo a su implementación. Se acompañará el avance del proyecto con el diseño de un intérprete adecuado. En el escrito nos concentraremos principalmente en el estudio teórico del lenguaje, donde solo se comentarán algunos detalles sobre su implementación. A lo largo del trabajo se empleará el nombre tentativo ΔΔLang para hacer referencia al lenguaje.

1.1. Motivación

En la materia *Algoritmos y Estructura de Datos II*, de *FaMAF*, se utiliza desde hace años un pseudocódigo para la enseñanza de conceptos tales como análisis de algoritmos, definición de tipos abstractos de datos, o la comprensión de distintas técnicas de programación. Este pseudocódigo está inspirado en el lenguaje imperativo *Pascal*, diseñado por *Niklaus Wirth* cerca de 1970 [1], al cual se le han ido realizando modificaciones de acuerdo a las necesidades didácticas de la materia. Algunas características que se han mantenido comprenden la sintaxis verbosa, el tipado fuerte para expresiones, y el formato estructurado del código. Haciendo un análisis exhaustivo de las clases y los ejemplos en los que se utiliza el lenguaje en la materia, observamos que su definición aún no está completamente clara, y que el grado de precisión con el que se declaran los programas varía entre una presentación y otra.

```
1 { PRE: 1 <= i, j <= n }
2 proc swap ( in / out a : array [1..n] of T, in i, j : nat )
3   var temp : T
4   temp := a[i]
5   a[i] := a[j]
6   a[j] := temp
7 end proc
```

Código 1.1: Permutación de Valores

Mediante los procedimientos `swap` y `selectionSort`, declarados en el pseudocódigo de la materia, se define un algoritmo para ordenar arreglos de menor a mayor. Cada uno de los pasos ha efectuar por el algoritmo es detallado con precisión, por lo que es sencillo imaginar una posible ejecución del programa dado con un arreglo `a` cualquiera. La traducción del código a un lenguaje de programación imperativo, como por ejemplo *C*, resulta ser casi directa.

```
proc selectionSort ( in/out a : array [1..n] of T )
  var minP : nat
  for i := 1 to n do
    minP := i
    for j := i + 1 to n do
      if a[j] < a[minP] then
        minP := j
      fi
    od
    swap(a, i, minP)
  od
end proc
```

Código 1.2: Ordenación por Selección

Con la función `cambio` se busca resolver el *problema de la moneda*, donde se intenta pagar un monto de dinero `m` con la menor cantidad de monedas posibles cuyas denominaciones válidas se encuentran en el conjunto `C`; el cual tiene una cantidad infinita de cada una. A diferencia del ejemplo previo ciertos pasos del algoritmo resultan ambiguos, donde en el pseudocódigo se permite escribir una oración en *lenguaje natural* para describir la acción a realizar durante su ejecución. Un problema que conlleva esto para un programador no experimentado es responder a la pregunta *¿Cuál es el grado de ambigüedad permitido?*

```
fun cambio ( m : monto, C : Conjunto de Monedas )
ret S : Conjunto de Monedas
  var c, resto : monto
  S := {} { Inicializamos S con el conjunto vacío }
  resto := m
  while resto > 0 do
    c := mayor elemento de C
    C := C - {c}
    S := S ∪ { resto div c monedas de denominación c }
    resto := resto mod c
  od
end fun
```

Código 1.3: Problema de la Moneda

En este trabajo nos proponemos definir con precisión un lenguaje como el que se utiliza en la materia, que permita al estudiante poder implementar y ejecutar los algoritmos presentados en el teórico y en el práctico. Además del beneficio desde el punto de vista didáctico, la definición formal e implementación de un lenguaje con estas características reviste interés propio. En el mismo se utilizan algunos conceptos avanzados de lenguajes de programación como el polimorfismo paramétrico, una variante primitiva de polimorfismo *ad hoc*, el manejo de memoria dinámica mediante punteros, pasaje por valor y referencia a través de funciones y procedimientos, entre otros. Para lograr el objetivo propuesto debemos primero definir precisamente la sintaxis del lenguaje, identificar las ambigüedades que naturalmente tiene debido al uso didáctico con el que fue creado, luego definir chequeos estáticos como el tipado, el alcance de variables, atributos sobre parámetros de funciones y procedimientos, entre otros.

1.2. Conceptos

A continuación introduciremos algunos conceptos generales que muy probablemente aparecen en el estudio y definición formal de cualquier lenguaje de programación, y por lo tanto también en este trabajo puntual. En particular nos enfocaremos en la sintaxis, concreta y abstracta, y los chequeos estáticos.

La **sintaxis concreta** define qué cadenas de caracteres serán consideradas programas del lenguaje. Los programas que uno finalmente escribe son caracterizados por esta gramática, la cual por lo general consiste de un conjunto de reglas que definen la manera de declarar programas de acuerdo a la *gramática libre de contexto* del lenguaje. Notar que si la gramática define como mencionamos qué programas están correctamente escritos, entonces es porque contempla detalles precisos sobre la escritura de las construcciones del lenguaje. Tales como los caracteres válidos para nombrar variables, las llaves para delimitar bloques de código, o las palabras claves correspondientes para denotar determinadas construcciones. Si por ejemplo consideramos al lenguaje de programación C, podríamos definir un pequeño fragmento de su sintaxis concreta para la declaración de funciones.

$$\begin{aligned}
\langle concreteFun \rangle &::= \langle type \rangle\ \langle id \rangle\ (\ \langle optionalParams \rangle\ )\ \{\ \langle body \rangle\ \} \\
\langle optionalParams \rangle &::= \langle parameters \rangle \mid \epsilon \\
\langle parameters \rangle &::= \langle parameters \rangle, \langle parameter \rangle \mid \langle parameter \rangle \\
\langle parameter \rangle &::= \langle id \rangle : \langle type \rangle
\end{aligned}$$

Asumiendo tenemos definidas las gramáticas para el resto de las construcciones no terminales $\langle type \rangle$, $\langle id \rangle$, y $\langle body \rangle$, notar que escribir una función en C consiste en escribir el tipo de retorno, su nombre seguido de los parámetros que necesariamente deben estar encerrados entre paréntesis y separados por comas, y finalmente entre llaves escribimos el cuerpo de la función. Observar lo engorrosa que resulta la definición de la gramática para los parámetros de la función, debido que es necesario caracterizar con precisión sus posibles variantes. Por ejemplo, un fragmento de la implementación de `factorial` podría ser de la siguiente forma.

$$\texttt{int factorial (n : int) } \{\ body_{conc}\ \}$$

Todos estos detalles burocráticos de escritura son importantes ya que, si bien algunas veces la sobrecargan, ayudan entre otras cosas a la lectura y organización del código. Sin embargo, una representación más abstracta de las componentes sintácticas que definen un programa suele resultar más conveniente para el estudio formal de propiedades como el chequeo de tipos, entre otras. Así como también una representación interna de un programa en la implementación de un compilador o intérprete.

La **sintaxis abstracta** define de manera independiente de cualquier representación particular a las *frases abstractas* que conforman a un lenguaje. Definiendo una *gramática abstracta* nos enfocamos solo en la información que nos resulta importante de una frase, omitiendo por lo tanto detalles de su representación específica. Una posible representación abstracta de las funciones de C es la siguiente, donde la declaración de una función (en su versión abstracta) tendrá que tener un nombre, sus argumentos, un tipo de retorno, y el cuerpo; omitiendo por ejemplo la escritura concreta de llaves, e incluso el orden en que aparecen las componentes puede resultar irrelevante. Es importante aclarar que los no terminales que no poseen producciones, son definidos de manera abstracta de forma análoga a los asumidos en la sintaxis concreta.

$$\langle abstractFun \rangle ::= \texttt{CFun}\ \langle id \rangle\ (\langle id \rangle, \langle type \rangle)_1 \ldots (\langle id \rangle, \langle type \rangle)_n\ \langle type \rangle\ \langle body \rangle$$

Finalmente declaradas ambas sintaxis podemos definir una función de traducción de la concreta a la abstracta, comúnmente denominada **parser**. Al aplicar la traducción a la función `factorial` que escribimos anteriormente, obtenemos la siguiente versión abstracta.

$$\texttt{CFun factorial (n, int) int } body_{abs}$$

Claramente no cualquier cadena de caracteres conforma un programa *bien escrito*, esto ocurre cuando al aplicar la función de traducción esta nos construye una frase abstracta. Sin embargo la construcción exitosa de la frase no significa que no existan errores; solo significa que el programa está correctamente escrito según las producciones que caracterizan a la sintaxis concreta. Todavía pueden existir errores como el uso de variables no declaradas, llamadas de funciones con una cantidad incorrecta de argumentos, o para el caso de lenguajes tipados, errores de tipo. Para capturar esta clase de errores definimos los **chequeos estáticos** que serán propiedades que le pediremos a un programa bien escrito y que podemos verificar sin ejecutarlo. Estos se definen sobre las frases abstractas aprovechando, como ya mencionamos, que solo cargan la información relevante de un programa concreto. Definiendo *juicios de tipado*, enunciados que establecen cuando una frase abstracta tiene un determinado tipo, podemos dar una prueba que establezca cuándo un programa no tiene errores de tipo; al poder asignarle uno. De igual manera es posible definir otras clases de enunciados para el resto de los chequeos.

Si chequeamos estáticamente la frase abstracta correspondiente con el ejemplo de la función `factorial`, deberíamos poder verificar que esta toma un entero y devuelve un entero, es decir que tiene tipo *int en int*. Es fundamental notar que al efectuar los chequeos sin ejecutar el programa, es posible que sigan existiendo errores que solo podrán ser capturados durante su ejecución, como la división por cero, o accesos de memoria no válidos, entre otros.

1.3. Organización de la Tesis

El escrito se divide en cinco capítulos. En el capítulo 2 se presenta el lenguaje, donde se ilustran sus características más relevantes a través de ejemplos. En el capítulo 3 se describe la sintaxis abstracta y se presenta la sintaxis concreta, además se detallan aspectos sobre la implementación del parser del lenguaje. En el capítulo 4 se definen formalmente los chequeos estáticos, y se lo acompaña con derivaciones de ejemplos. En el capítulo 5 se concluye el trabajo con un breve resumen de las actividades desarrolladas y un listado de trabajos futuros.

Capítulo 2

Sobre el lenguaje

ΔΔLang es un lenguaje de programación imperativo estructurado fuertemente tipado, inspirado en *Pascal*, sobre el cual se han agregado características avanzadas como polimorfismo paramétrico, y una versión básica de polimorfismo *ad hoc*. En este capítulo, nos concentraremos en exponer sus aspectos principales a través de ejemplos, de manera que resulte como un breve tutorial de uso para el lector familiarizado con la programación imperativa. Al encontrarse aún en etapa de desarrollo, ciertos aspectos del lenguaje se encuentran sujetos a posibles modificaciones en futuras iteraciones.

2.1. Tipos Nativos

El lenguaje posee tipado fuerte, por lo que toda expresión tiene asociado un tipo de datos particular. De manera nativa, se encuentran definidos algunos tipos los cuales serán divididos en básicos y estructurados.

2.1.1. Tipos Básicos

Un tipo básico representa un conjunto de valores individuales. En el lenguaje se encuentran definidos los tipos numéricos, el tipo booleano, y el tipo de los caracteres. Para cada uno existe un conjunto de constantes que representa a sus valores, y algunos operadores que permiten manipular expresiones.

- Para representar los valores numéricos, el lenguaje define los tipos `int` y `real`. Los operadores aritméticos clásicos se encuentran sobrecargados para operar con valores de ambos tipos de datos.

```
1 var i : int
2 var r : real
3 i := 10 / 2
4 r := 5.0 + 4.5
```

  Una expresión de tipo entero puede ser utilizada en contextos donde se espera una expresión de tipo real. De esta manera es posible tener expresiones donde se emplean valores numéricos de ambos tipos. En el siguiente ejemplo se divide una sumatoria de números enteros, por un número entero, para asignar un valor a la variable real `average`.

```
var i1, i2, i3 : int
var average : real
...
average := (i1 + i2 + i3) / 3
```

- Los valores de verdad están representados mediante el tipo `bool`. Se definen las constantes `true` y `false`, junto con los operadores lógicos clásicos `&&`, `||`, y `!`. Además se encuentran definidos los operadores de comparación tradicionales.

```
var i : int
var b : bool
...
b := b && i < 10
```

- Los caracteres son representados por el tipo `char`. Se encuentran definidas cada una de las constantes `ASCII`. A diferencia de los tipos anteriores no tenemos definidas operaciones para este tipo de datos.

```
var c : char
c := 'z'
```

En los ejemplos previos, se ilustra la declaración y asignación de variables. Para la declaración utilizamos la palabra clave `var` seguida del identificador de la variable junto con su respectivo tipo. Para la asignación empleamos el operador `:=` para asignar el valor de una expresión particular a una variable determinada.

2.1.2. Tipos Estructurados

Un tipo estructurado permite representar colecciones de otros tipos de datos. De manera similar a los tipos básicos, se definen operaciones específicas para acceder a los elementos que conforman al tipo. En el lenguaje solo tenemos definidos de forma nativa a los arreglos.

Los arreglos, representados por el tipo `array`, permiten agrupar una cantidad fija de elementos de algún tipo de datos. Cada elemento se ubica en una posición determinada, designada por índices enteros. Para definir un arreglo es necesario detallar el tipo de sus componentes y los tamaños para cada una de sus dimensiones, los cuales deberán ser mayores a cero.

```
var a : array [3] of real
a[0] := 10.75
a[1] := 5.5
a[2] := 20.25
```

En el ejemplo, se declara un arreglo unidimensional de reales `a` con solo tres elementos. Para acceder a una posición determinada, se utilizan corchetes `[]` junto con el índice correspondiente. Los índices válidos comienzan desde el cero, y continúan hasta el tamaño de la respectiva dimensión sin incluirlo.

Es posible crear arreglos con múltiples dimensiones especificando el tamaño de cada una en el momento de su declaración. En el siguiente ejemplo se define un arreglo bidimensional `a` con ambas dimensiones de tamaño cinco. Para acceder a una posición determinada, simplemente se separan con comas `,` las correspondientes coordenadas.

```
var a : array [5, 5] of real
for i := 0 to 4 do
  for j := 0 to 4 do
    a[i, j] := 0.0
  od
od
```

Notar que se inicializan todas las posiciones del arreglo, utilizando la sentencia `for`. La declaración de las variables `i` y `j`, es implícita; donde su tipo es inferido de acuerdo a las expresiones especificadas como rangos de la iteración, y su alcance comprende el cuerpo de la sentencia.

En ciertas situaciones, se permiten emplear identificadores para representar el tamaño de algunas dimensiones de un arreglo. De esta manera es posible abstraernos de sus tamaños concretos en el código, donde es importante no confundir esta característica que nos permite abstraer el *tamaño fijo* de un arreglo, con la posibilidad de definir arreglos con dimensiones de *tamaño dinámico*; lo cual no se permite en el lenguaje. En el ejemplo se utiliza el tamaño `n` como una constante, la cual debe haber sido introducida previamente en el código, y cuyo valor será resuelto durante la ejecución del programa.

```
var a : array [n] of real
for i := 0 to n - 1 do
  a[i] := 0.0
od
```

2.2. Definición de Tipos

En el lenguaje podemos extender el conjunto de tipos de datos, definiendo nuevos tipos. Para ello declaramos mediante la palabra clave `type` seguida de un nombre el nuevo tipo de datos, que podrá construirse de tres formas distintas.

2.2.1. Enumerados

Un tipo enumerado representa un conjunto finito de valores. Cada valor está definido mediante un identificador único. Para declarar un tipo enumerado se emplean las palabras claves `enumerate` y `end enumerate`.

```
type day = enumerate
           Sunday
           Monday
           Tuesday
           Wednesday
           Thursday
           Friday
           Saturday
           end enumerate
```

Declarado un tipo enumerado, se permiten emplear sus valores como constantes definidas. En el siguiente ejemplo, se inicializa la variable `d` con la constante `Sunday`.

```
var d : day
d := Sunday
```

2.2.2. Sinónimos

Un sinónimo de tipo es un *renombre* de un tipo existente. En su declaración solo se requiere detallar el tipo asociado, lo cual permite utilizar este nuevo nombre para el mismo.

```
type matrixReal = array [5, 5] of real
```

Una expresión de cierto tipo de datos puede ser empleada en contextos donde se espera un valor de uno de los sinónimos de su tipo. En el siguiente ejemplo se declara una variable `mR` del tipo `matrixReal`, y se opera de manera transparente como si fuese un arreglo tradicional.

```
var mR : matrixReal
for i := 0 to 4 do
  for j := 0 to 4 do
    mR[i, j] := 0.0
  od
od
```

2.2.3. Tuplas

Una tupla es una colección finita de campos, posiblemente de diferentes tipos de datos, donde cada uno tiene asociado un identificador. Las tuplas son utilizadas para agrupar un conjunto de valores que se relacionan de alguna manera. Para su declaración se emplean las palabras claves `tuple` y `end tuple`. Con la operación `.` se accede al campo de una tupla, de acuerdo a un identificador determinado.

```
type person = tuple
              initial : char,
              age     : int,
              weight  : real
              end tuple
```

En el ejemplo anterior se definió el tipo `person` mediante una tupla con los campos `initial`, `age`, y `weight`, donde sus respectivos tipos son `char`, `int`, y `real`. En el ejemplo siguiente se declara una variable p del tipo introducido, y se inicializan todos los campos de manera adecuada.

```
var p : person
p.initial := 'F'
p.age     := 30
p.weight  := 70.5
```

2.3. Funciones y Procedimientos

La única manera de escribir programas en nuestro lenguaje es mediante la declaración de funciones y procedimientos. En esencia estas construcciones comprenden un bloque de código estructurado, implementado para realizar una tarea particular.

Una función realiza una computación de acuerdo a un conjunto de parámetros, y al finalizar retorna siempre un resultado en el contexto donde fue llamada. Su comportamiento es determinado solamente por los valores de los argumentos que recibe, donde tampoco podrá modificar el estado de los mismos.

```
{- PRE: n >= 0 -}
fun factorial ( n : int ) ret fact : int
  fact := 1
  for i := 2 to n do
    fact := fact * i
  od
end fun
```

En el ejemplo se muestra la implementación de la función `factorial`, que calcula el factorial de un número entero positivo `n`. La variable de retorno `fact` almacena la productoria de números, y al finalizar la ejecución de la función, se retorna su valor al contexto donde se efectuó la llamada. El comentario simplemente indica la precondición ha satisfacer para garantizar el comportamiento esperado de la función.

Un procedimiento realiza una computación de acuerdo a un conjunto de parámetros de lectura, para modificar un conjunto de parámetros de escritura. Su comportamiento es determinado solamente por los parámetros que recibe donde cada uno lleva un decorado que indica si es de lectura `in`, de escritura `out`, o ambas `in/out`. Un procedimiento no modifica el estado de los parámetros de lectura, y tampoco consulta el estado de los parámetros de escritura.

```
proc initialize ( in e : int, out a : array [10] of int )
  for i := 9 downto 0 do
    a[i] := e
  od
end proc
```

En el ejemplo se implementa el procedimiento `initialize`, que inicializa un arreglo de enteros de acuerdo a un valor determinado. Notar que el parámetro de lectura `e` solo ocurre del lado derecho de la asignación, mientras que el parámetro de escritura `a` solo ocurre del lado izquierdo.

Previamente cuando se presentó el tipo de los arreglos, se mencionó que el lenguaje permite la utilización de identificadores para describir que un arreglo tiene un tamaño fijo que aun no se ha determinado (lo cual no debe confundirse con arreglos de tamaños dinámicos que el lenguaje no posee). La introducción de tales identificadores solo es posible durante la declaración del prototipo de una función o un procedimiento, y siempre en el contexto de un parámetro de tipo `array`. Adicionalmente este identificador podrá ser utilizado como una constante en el respectivo cuerpo de la construcción declarada.

```
{- PRE: 0 <= i, j < n -}
proc swap ( in/out a : array [n] of int, in i, j : int )
  var temp : int
  temp := a[i]
  a[i] := a[j]
  a[j] := temp
end proc
```

En el ejemplo se declara el procedimiento `swap`, que intercambia los valores de las posiciones `i` y `j` del arreglo `a`. Notar que el parámetro `a` de tipo `array` introduce el identificador `n`, lo cual nos permite abstraer el tamaño fijo del arreglo. En consecuencia, se generaliza la declaración del procedimiento ya que podrá ser aplicado a arreglos de una dimensión con cualquier tamaño; a diferencia de si en lugar de `n` se hubiese precisado algún tamaño puntual utilizando un entero. Es importante notar que aunque el procedimiento puede ser aplicado a arreglos de cualquier tamaño, estos serán fijos. Lo cual significa que el tamaño del arreglo `a` no debe ser pensado como

dinámico, sino como un tamaño fijo `n` aun no determinado; lo cual ocurrirá al momento de la aplicación del procedimiento.

El lenguaje soporta recursión. La declaración de funciones y procedimientos permite definiciones recursivas, donde en el respectivo cuerpo es posible llamar a la construcción que está siendo definida. En el siguiente ejemplo se muestra una implementación alternativa para la función `factorial` utilizando recursión.

```
{- PRE: n >= 0 -}
fun factorial ( n : int ) ret fact : int
  if n >= 2 then
    fact := n * factorial(n - 1)
  else
    fact := 1
  fi
end fun
```

El orden de declaración de funciones y procedimientos es fundamental, ya que solo se permite efectuar una llamada a funciones o procedimientos que hayan sido definidos con anterioridad. En particular, no es posible definir funciones o procedimientos mediante recursión mutua.

2.4. Polimorfismo Paramétrico

El polimorfismo es la característica de los lenguajes de programación mediante la cual se pueden definir funciones y procedimientos de manera genérica para diferentes tipos de datos. Consideremos los ejemplos previos de inicializar un arreglo con un elemento dado, y de intercambiar posiciones de elementos en un arreglo. En las definiciones anteriores el arreglo es de enteros. Sin embargo las tareas de inicializar o intercambiar elementos también pueden ser útiles en arreglos de otros tipos de datos. Podríamos entonces redefinir ambos procedimientos con los nombres `initializeInt`, `swapInt`, y luego definir procedimientos *idénticos* a estos, pero ahora recibiendo arreglos de booleanos, llamándolos por ejemplo `initializeBool`, `swapBool`.

```
proc initializeInt ( in e : int, out a : array [n] of int )
  for i := n - 1 downto 0 do
    a[i] := e
  od
end proc
```

```
proc initializeBool ( in e : bool, out a : array [n] of bool )
  for i := n - 1 downto 0 do
    a[i] := e
  od
end proc
```

```
{- PRE: 0 <= i, j < n -}
proc swapInt ( in/out a : array [n] of int, in i, j : int )
  var temp : int
  temp := a[i]
  a[i] := a[j]
  a[j] := temp
end proc
```

```
{- PRE: 0 <= i, j < n -}
proc swapBool ( in/out a : array [n] of bool, in i, j : int )
  var temp : bool
  temp := a[i]
  a[i] := a[j]
  a[j] := temp
end proc
```

Si luego quisiéramos implementar la misma tarea pero para otro tipo de datos, deberíamos volver a escribir una vez más el mismo código. El *polimorfismo paramétrico* permite definir una única vez estos procedimientos de manera genérica, permitiendo que sean utilizados para cualquier tipo de datos.

```
proc initialize ( in e : T, out a : array [n] of T )
  for i := n - 1 downto 0 do
    a[i] := e
  od
end proc
```

```
{- PRE: 0 <= i, j < n -}
proc swap ( in/out a : array [n] of T, in i, j : int )
  var temp : T
  temp := a[i]
  a[i] := a[j]
  a[j] := temp
end proc
```

Notemos que las tareas de inicializar e intercambiar son independientes del tipo de datos almacenado por el arreglo. La tarea de inicializar recorre cada posición del arreglo y almacena el valor dado, y la tarea de intercambiar almacena los correspondientes valores en las posiciones intercambiadas; sea cual sea el tipo de datos de los elementos del arreglo ambas tareas son idénticas. La característica que permite definir una *misma* función o procedimiento para cualquier tipo de datos, se denomina **polimorfismo paramétrico**.

La idea de la declaración de funciones y procedimientos cuyos comportamientos son independientes del tipo de datos a los que se aplican, puede ser adaptada para la declaración de tipos. Consideremos el siguiente ejemplo donde se define el tipo tupla `pair`. La declaración de un tipo paramétrico nos permite generalizar su estructura, la cual será independiente de los tipos que serán instanciados en sus parámetros.

```
type pair of (A, B) = tuple
                        first  : A,
                        second : B
                        end tuple
```

El polimorfismo paramétrico permite definir una única vez el tipo, que luego podrá ser instanciado de manera apropiada según sea necesario. Si estuviésemos trabajando con puntos en un plano, por ejemplo, una forma para representarlos sería mediante el tipo `pair of (real,real)` donde cada campo corresponde a una coordenada del punto respectiva a los ejes del plano.

```
fun first (p : pair of (A, B)) ret fst : A
  fst := p.first
end fun
```

```
fun second (p : pair of (A, B)) ret snd : B
  snd := p.second
end fun
```

Combinando los conceptos presentados sobre el polimorfismo paramétrico, podemos implementar los ejemplos anteriores `first` y `second`. La operación de obtener cualquiera de los campos de la tupla, es independiente del tipo de datos almacenados en la misma. Notar que ambos conceptos se complementan de cierta manera, donde funciones y procedimientos polimórficos pueden hacer uso de los tipos paramétricos.

2.5. Polimorfismo *Ad Hoc*

En muchas ocasiones podemos identificar que una función o un procedimiento podrían ser implementados de manera genérica pero no para cualquier tipo de datos, sino para ciertos tipos que comparten alguna característica, en consecuencia no es posible utilizar *polimorfismo paramétrico*. Consideremos las siguientes implementaciones de `belongs` y `selectionSort`. La primera decide si un valor determinado pertenece a un arreglo y la segunda permite ordenar un arreglo de menor a mayor.

```
fun belongs ( e : int, a : array [n] of int ) ret b : bool
  var i : int
  i := 0
  b := false
  while !b && i < n do
    b := a[i] == e
    i := i + 1
  od
end fun
```

```
proc selectionSort ( in/out a : array [n] of int )
  var minPos : int
  for i := 0 to n - 1 do
    minPos := i
    for j := i + 1 to n - 1 do
      if a[j] < a[minPos] then minPos := j fi
    od
    swap(a, i, minPos)
  od
end proc
```

En ambas implementaciones los elementos son de tipo entero. Sin embargo podríamos definir las mismas operaciones para otros tipos de datos, como por ejemplo, caracteres. Se tendrían que redefinir las anteriores con los nombres `belongsInt` y `selectionSortInt`, y declarar de manera idéntica las operaciones `belongsChar` y `selectionSortChar` donde solo cambiaríamos `int` por `char` en los tipos de los parámetros. Con un trabajo tediosamente repetitivo se podrían dar declaraciones para todos los tipos que tengan definidas las operaciones de comparación; aunque no sería posible para aquellos que no las tengan definidas.

El *polimorfismo ad hoc* nos permite escribir de manera genérica una función o un procedimiento donde la tarea que realiza sólo está bien definida para algunos tipos. Además esta tarea puede no ser la misma dependiendo del tipo.

En el lenguaje se definen una serie de clases las cuales pueden ser pensadas como una especie de interfaz que caracteriza algún comportamiento. Un tipo es una instancia de una clase, cuando implementa el comportamiento que la clase describe. El lenguaje sólo incorpora de forma nativa las clases `Eq` y `Ord`, y no existe posibilidad de declarar nuevas clases. La primera representa a los tipos que tienen alguna noción de igualdad, y sus operaciones definidas comprenden al `==` y `!=`. La segunda representa a los tipos que poseen alguna relación de orden, y sus operaciones definidas comprenden al `<`, `<=`, `>=`, `>`.

```
fun belongs ( e : T, a : array [n] of T ) ret b : bool
where (T : Eq)
  var i : int
  i := 0
  b := false
  while !b && i < n do
    b := a[i] == e
    i := i + 1
  od
end fun
```

```
proc selectionSort ( in/out a : array [n] of T )
where (T : Ord)
  var minPos : int
  for i := 0 to n - 1 do
    minPos := i
    for j := i + 1 to n - 1 do
      if a[j] < a[minPos] then minPos := j fi
    od
    swap(a, i, minPos)
  od
end proc
```

En los ejemplos, notemos que las tareas de comprobar pertenencia y ordenar elementos solo dependen de las operaciones de comparación soportadas por el tipo de datos almacenado por el arreglo. La función que comprueba la pertenencia recorre cada posición del arreglo, y aplica la operación `==` para verificar si los elementos coinciden, por lo tanto el tipo de los elementos del arreglo deberá tener instancia de `Eq`. El procedimiento que ordena los elementos recorre el arreglo intercambiando las posiciones de sus valores, de acuerdo al resultado de la operación `<`, es decir el tipo de los elementos deberá tener instancia de `Ord`.

2.6. Memoria Dinámica

El lenguaje permite manipular explícitamente la *memoria dinámica* mediante un tipo de datos especial, que llamaremos puntero. Supongamos que deseamos definir el tipo correspondiente a las listas. Una lista permite representar una colección ordenada de elementos de algún tipo de datos, cuyo tamaño es variable; lo cual significa que su tamaño crece tanto como sea necesario, de acuerdo a la cantidad de elementos almacenados. Todos los tipos presentados hasta el momento utilizan una cantidad fija de memoria, la cual no puede ser modificada en tiempo de ejecución. Recordemos una vez más que los arreglos implementados en el lenguaje tienen un tamaño fijo al momento de la ejecución. En este aspecto el uso de punteros resulta fundamental, ya que

permiten reservar y liberar memoria en la medida que sea necesario durante la ejecución del programa.

```
type node of (T) = tuple
                     elem : T,
                     next : pointer of node of (T)
                     end tuple

type list of (T) = pointer of node of (T)
```

Un puntero, representado por el tipo `pointer`, indica el lugar en memoria donde se aloja un elemento de cierto tipo. Combinando la declaración de tipos utilizando tuplas con punteros, se permiten declarar definiciones recursivas en el lenguaje; lo cual resulta indispensable para la declaración de las listas. En el ejemplo anterior se declara una *lista enlazada* denominada `list`, la cual se compone de una sucesión de nodos `node` que se integran por los campos `elem` de cierto tipo paramétrico `T`, y `next` el cual referencia al siguiente nodo en la lista.

```
var p : pointer of int
alloc(p)
...
#p := #p + 1
...
free(p)
```

En el lenguaje se definen tres operaciones para manipular punteros. El procedimiento nativo `alloc` toma una variable de tipo puntero, y le asigna la dirección de un nuevo bloque de memoria, cuyo tamaño estará determinado por el tipo de la variable. El operador `#` permite acceder al bloque de memoria apuntado por el puntero. El procedimiento nativo `free` toma una variable de tipo puntero, y libera el respectivo bloque de memoria referenciado.

Retomando el ejemplo de la *lista enlazada*, los procedimientos `empty` y `addL` son utilizados para construir valores del tipo en cuestión.

```
proc empty ( out l : list of (T) )
  l := null
end proc
```

El procedimiento `empty` construye una lista vacía. La constante `null` representa un puntero que no referencia un lugar de memoria válido. En el ejemplo, la constante representa una lista que no posee ningún nodo.

```
proc addL ( in e : T, in/out l : list of (T) )
  var p : pointer of node of (T)
  alloc(p)
  p->elem := e
  p->next := l
  l := p
end proc
```

El procedimiento `addL` agrega un elemento al principio de la lista. Se reserva memoria para un nodo, se inicializan sus campos, y se modifica la lista para que señale al elemento en cuestión. El operador `->` es una notación conveniente para acceder al campo de una tupla señalada por un puntero; lo cual evita tener que emplear ambos operadores de acceso a punteros y tuplas, en lugar de escribir `#p.elem` simplemente se denota `p->elem` para acceder a un elemento de la lista.

Capítulo 3

Sobre la sintaxis

En el siguiente capítulo, nos dedicaremos principalmente a la definición del lenguaje. Se precisará de manera formal su sintaxis abstracta, y se presentará de manera informal su sintaxis concreta. Una vez definidas, comenzaremos con la descripción de los aspectos principales sobre la implementación del parser de nuestro lenguaje.

3.1. Sintaxis Abstracta

La sintaxis del lenguaje ya se expuso de manera informal en el capítulo anterior, mediante los distintos ejemplos que se fueron ilustrando a lo largo de su desarrollo. A continuación describiremos de manera formal la *sintaxis abstracta* de ΔΔLang, donde ya mencionamos que esta sintaxis pretende capturar las componentes relevantes de las distintas construcciones sintácticas del lenguaje eliminando detalles poco importantes de su sintaxis concreta.

3.1.1. Expresiones

Una expresión puede adoptar distintas formas; puede ser un valor constante, una llamada a función, una operación sobre otras expresiones, o una variable con sus respectivos operadores. Su composición se describe a continuación.

$\langle expression \rangle$ ::= $\langle constant \rangle$ | $\langle functioncall \rangle$ | $\langle operation \rangle$ | $\langle variable \rangle$

Una constante puede pertenecer a alguna de las siguientes clases de valores. Los no terminales $\langle integer \rangle$, $\langle real \rangle$, $\langle bool \rangle$, y $\langle character \rangle$ denotan los conjuntos de valores esperados, mientras que $\langle cname \rangle$ hace referencia a los identificadores de constantes enumeradas definidas por el usuario. Los terminales **inf** y **null**, representan al infinito y al puntero nulo respectivamente.

$\langle constant \rangle$ ::= $\langle integer \rangle$ | $\langle real \rangle$ | $\langle bool \rangle$ | $\langle character \rangle$ | $\langle cname \rangle$ | **inf** | **null**

Una llamada a función está compuesta por su nombre, y la lista de argumentos que recibe; la cual puede tener una cantidad arbitraria de parámetros. Notar que se utilizará la misma clase de identificadores tanto para funciones y procedimientos, como para variables.

$\langle functioncall \rangle$ ::= $\langle id \rangle$ ($\langle expression \rangle$ ... $\langle expression \rangle$)

Los operadores del lenguaje están conformados por las operaciones numéricas y booleanas tradicionales, y por los operadores de orden e igualdad respectivos a las clases definidas. Observar que será necesario la implementación de un chequeo de tipos para asegurar su uso apropiado.

⟨*operation*⟩ ::= ⟨*expression*⟩ ⟨*binary*⟩ ⟨*expression*⟩ | ⟨*unary*⟩ ⟨*expression*⟩

⟨*binary*⟩ ::= + | − | ∗ | / | % | || | && | <= | >= | < | > | == | !=

⟨*unary*⟩ ::= − | !

Finalizando con las expresiones, describiremos a las variables con sus respectivos operadores. Una variable puede simbolizar un único valor, un arreglo de varias dimensiones, una tupla con múltiples campos, o un puntero a otra estructura en memoria. El no terminal ⟨$fname$⟩ representa el nombre de un campo de una estructura de tipo tupla definida por el usuario.

⟨*variable*⟩ ::= ⟨*id*⟩
| ⟨*variable*⟩ [⟨*expression*⟩ ... ⟨*expression*⟩]
| ⟨*variable*⟩ . ⟨$fname$⟩
| ⋆ ⟨*variable*⟩

3.1.2. Sentencias

Las sentencias del lenguaje se dividen en las siguientes construcciones. La composición de la *asignación* y el *while* es bastante simple, por lo que se detallan también a continuación. Notar que el no terminal ⟨*sentences*⟩ se utiliza para representar al bloque de sentencias.

⟨*sentence*⟩ ::= **skip** | ⟨*assignment*⟩ | ⟨*procedurecall*⟩ | ⟨if⟩ | ⟨*while*⟩ | ⟨for⟩

⟨*assignment*⟩ ::= ⟨*variable*⟩ := ⟨*expression*⟩

⟨*while*⟩ ::= **while** ⟨*expression*⟩ **do** ⟨*sentences*⟩

⟨*sentences*⟩ ::= ⟨*sentence*⟩ ... ⟨*sentence*⟩

Para la llamada de procedimientos, se utiliza una sintaxis idéntica a la empleada para la llamada de funciones. Adicionalmente se encuentran definidos los procedimientos especiales **alloc** y **free**, exclusivos para el manejo dinámico de memoria del programa.

⟨*procedurecall*⟩ ::= ⟨*id*⟩ (⟨*expression*⟩ ... ⟨*expression*⟩)
| **alloc** ⟨*variable*⟩
| **free** ⟨*variable*⟩

La especificación de la sentencia *if* es compleja en lo que refiere a su sintaxis concreta. En cambio para obtener una sintaxis abstracta simple, nos limitaremos a solo permitir una única alternativa para su especificación.

⟨if⟩ ::= **if** ⟨*expression*⟩ **then** ⟨*sentences*⟩ **else** ⟨*sentences*⟩

Finalizando con las sentencias, otra instrucción que presenta varias alternativas es el *for*. Con la sentencia es posible declarar una variable, la cual tomará valores en un rango determinado, y ejecutar de forma iterada un bloque de sentencias específico. Los rangos ascendentes se definen con **to**, y los rangos descendentes con **downto**.

⟨for⟩ ::= **for** ⟨*id*⟩ := ⟨*expression*⟩ **to** ⟨*expression*⟩ **do** ⟨*sentences*⟩
| **for** ⟨*id*⟩ := ⟨*expression*⟩ **downto** ⟨*expression*⟩ **do** ⟨*sentences*⟩

3.1.3. Tipos

Los tipos que soporta ΔΔLang pueden dividirse en dos categorías, los nativos del lenguaje y los definidos por el usuario. A su vez los nativos pueden separarse en básicos (**int**, **real**, **bool**, **char**), estructurados ($\langle array \rangle$), y punteros ($\langle pointer \rangle$).

$\langle type \rangle$ ::= **int** | **real** | **bool** | **char**
| $\langle array \rangle$
| $\langle pointer \rangle$
| $\langle definedtype \rangle$
| $\langle typevariable \rangle$

Del lado de los tipos nativos restantes, se tienen a los arreglos y a los punteros. Para los primeros, hay que especificar como se definen los tamaños para sus dimensiones. El no terminal $\langle sname \rangle$ representa a los identificadores empleados para abstraer los tamaños de arreglos, los cuales son introducidos en el prototipo de funciones y procedimientos. Para los segundos, solo se debe indicar cual es el tipo de valores que serán señalados por el puntero.

$\langle array \rangle$ ::= **array** $\langle arraysize \rangle$ ... $\langle arraysize \rangle$ **of** $\langle type \rangle$

$\langle arraysize \rangle$ ::= $\langle natural \rangle$ | $\langle sname \rangle$

$\langle pointer \rangle$ ::= **pointer** $\langle type \rangle$

En el caso de las variables de tipo, las mismas poseen su propia clase de identificadores específicos. En cambio para los tipos definidos por el usuario, además de su nombre representado por el no terminal $\langle tname \rangle$, se deberán detallar los tipos en los cuales es instanciado. Si el mismo no posee parámetros, el terminal **of** podrá ser obviado para abreviar la notación.

$\langle typevariable \rangle$ ::= $\langle typeid \rangle$

$\langle definedtype \rangle$ ::= $\langle tname \rangle$ **of** $\langle type \rangle$ ... $\langle type \rangle$

Cuando se declara un procedimiento, es necesario especificar el rol que cumplirá cada uno de sus parámetros. Lo cual significa que se debe detallar, para todos los parámetros individualmente, si se emplearán para lectura (**in**), escritura (**out**), o ambas (**in/out**).

$\langle io \rangle$ ::= **in** | **out** | **in/out**

Existen una serie de clases predefinidas para los tipos del lenguaje, donde cada clase representa una especie de interfaz que caracteriza las propiedades que cumplen cada uno de los tipos que las definen. En la versión actual del lenguaje solo se precisan formalmente dos clases, la primera para comprobar igualdad de elementos y la segunda para ordenar valores.

$\langle class \rangle$ ::= **Eq** | **Ord**

Para la declaración de un nuevo tipo por parte del usuario, existen tres posibilidades. En el lenguaje se pueden definir tipos enumerados, sinónimos de tipos, y estructuras de tipo tupla. Exceptuando al primero, al declarar un tipo es posible especificar parámetros de tipo que permiten crear construcciones más abstractas, aprovechando el polimorfismo paramétrico soportado por el lenguaje. Similar a lo dicho previamente, si los tipos declarados no poseen argumentos entonces el terminal **of** se omite para abreviar la notación.

⟨*typedecl*⟩ ::= **enum** ⟨*tname*⟩ = ⟨*cname*⟩ ... ⟨*cname*⟩
| **syn** ⟨*tname*⟩ **of** ⟨*typearguments*⟩ = ⟨*type*⟩
| **tuple** ⟨*tname*⟩ **of** ⟨*typearguments*⟩ = ⟨*field*⟩ ... ⟨*field*⟩

⟨*typearguments*⟩ ::= ⟨*typevariable*⟩ ... ⟨*typevariable*⟩

⟨*field*⟩ ::= ⟨*fname*⟩ : ⟨*type*⟩

3.1.4. Programas

Para finalizar con la sintaxis abstracta del lenguaje, debemos describir como se especifica un programa con la misma. Un programa está compuesto por una serie de declaraciones de tipo, seguidas de una serie de declaraciones de funciones y/o procedimientos.

⟨*program*⟩ ::= ⟨*typedecl*⟩ ... ⟨*typedecl*⟩ ⟨*funprocdecl*⟩ ... ⟨*funprocdecl*⟩

⟨*funprocdecl*⟩ ::= ⟨*function*⟩ | ⟨*procedure*⟩

El cuerpo de una función o un procedimiento está conformado primero por una lista de declaraciones de variables, y segundo por una lista de sentencias. Para declarar una variable solo se tiene que especificar su identificador, junto con el tipo que posee. Observar que existe la posibilidad de definir múltiples variables en una sola declaración, y que no es posible declarar variables entre las sentencias del cuerpo.

⟨*body*⟩ ::= ⟨*variabledecl*⟩ ... ⟨*variabledecl*⟩ ⟨*sentences*⟩

⟨*variabledecl*⟩ ::= **var** ⟨*id*⟩ ... ⟨*id*⟩ : ⟨*type*⟩

Una función posee un identificador propio, una lista de argumentos, un retorno, y un bloque que conforma su cuerpo. Tanto para los argumentos, como para el retorno, solo se tienen que detallar sus identificadores junto con el tipo de datos que representarán.

⟨*function*⟩ ::= **fun** ⟨*id*⟩ (⟨*funargument*⟩ ... ⟨*funargument*⟩) **ret** ⟨*funreturn*⟩
where ⟨*constraints*⟩
in ⟨*body*⟩

⟨*funargument*⟩ ::= ⟨*id*⟩ : ⟨*type*⟩

⟨*funreturn*⟩ ::= ⟨*id*⟩ : ⟨*type*⟩

Un procedimiento posee una estructura muy similar a la de una función. Posee un identificador propio, una lista de parámetros, y un bloque que conforma su cuerpo. La declaración de un parámetro requiere de la especificación de su identificador, del tipo de valor que representará, y de la etiqueta que caracteriza su uso de *entrada/salida*.

⟨*procedure*⟩ ::= **proc** ⟨*id*⟩ (⟨*procargument*⟩ ... ⟨*procargument*⟩)
where ⟨*constraints*⟩
in ⟨*body*⟩

⟨*procargument*⟩ ::= ⟨*io*⟩ ⟨*id*⟩ : ⟨*type*⟩

Debido que existe la posibilidad de definir funciones y procedimientos polimórficos, es conveniente poder restringir el polimorfismo agregando restricciones a las variables de tipo involucradas. De esta manera se pueden crear funciones y procedimientos más abstractos, cuyas implementaciones abarquen una gran variedad de tipos, pero al mismo tiempo requerir que sean instancias de determinadas clases del lenguaje.

$\langle constraints \rangle \ ::= \ \langle constraint \rangle \ \ldots \ \langle constraint \rangle$

$\langle constraint \rangle \ ::= \ \langle typevariable \rangle \ : \ \langle class \rangle \ \ldots \ \langle class \rangle$

3.2. Sintaxis Concreta

La *sintaxis concreta* de ΔΔLang fue introducida de manera informal en la presentación del lenguaje en el capítulo previo, y debido que para su estudio nos limitaremos a la *sintaxis abstracta*, no entraremos demasiado en detalle en este aspecto. De todas maneras, consideramos importante mencionar algunas características que pueden no haber sido detalladas de forma clara y que impactaron en el diseño del parser.

3.2.1. Identificadores

En el lenguaje hay diversas categorías de identificadores. Las cuales pueden ser separadas en dos clases, las que comienzan con minúscula $\langle lower \rangle$, y las que comienzan con mayúscula $\langle upper \rangle$. Para el resto del cuerpo, se tiene la misma estructura $\langle rest \rangle$ en ambos casos. En la primera clase se encuentran los identificadores de variables, funciones y procedimientos, a los tamaños abstractos de arreglos, los alias para campos de tuplas, y por último los tipos definidos por el usuario. Mientras que la segunda clase está conformada por los identificadores para variables de tipo, y las constantes enumeradas.

$\langle id \rangle \ ::= \ \langle lower \rangle \ \langle rest \rangle$

$\langle sname \rangle \ ::= \ \langle lower \rangle \ \langle rest \rangle$

$\langle fname \rangle \ ::= \ \langle lower \rangle \ \langle rest \rangle$

$\langle tname \rangle \ ::= \ \langle lower \rangle \ \langle rest \rangle$

$\langle typeid \rangle \ ::= \ \langle upper \rangle \ \langle rest \rangle$

$\langle cname \rangle \ ::= \ \langle upper \rangle \ \langle rest \rangle$

Se puede observar que de acuerdo al contexto y al formato del identificador parseado, somos capaces de distinguir a que clase de elemento se está haciendo referencia en el código, a excepción de un caso particular. Dentro de una expresión, no es posible diferenciar al identificador de una variable del identificador empleado para abstraer el tamaño de un arreglo. Lo cual nos obliga a tener una precaución adicional a la hora de los chequeos estáticos para ser capaces de reconocer a cada uno.

A continuación describimos los distintos caracteres que pueden conformar un identificador del lenguaje. En el cuerpo se permiten emplear combinaciones de letras $\langle letter \rangle$, dígitos $\langle digit \rangle$, y otros símbolos $\langle other \rangle$. Para el resto de *componentes léxicos*, como los números o los caracteres literales, su estructura no presenta ninguna particularidad relevante por lo que serán omitidos.

$\langle rest \rangle$ ::= ($\langle letter \rangle$ | $\langle digit \rangle$ | $\langle other \rangle$)*

$\langle letter \rangle$ ::= $\langle lower \rangle$ | $\langle upper \rangle$

$\langle lower \rangle$ ::= a | b | ... | z

$\langle upper \rangle$ ::= A | B | ... | z

$\langle digit \rangle$ ::= 0 | 1 | ... | 9

$\langle other \rangle$ ::= _ | $'$

3.2.2. Azúcar Sintáctico

Existen una serie de construcciones concretas en ΔΔLang que son definidas mediante azúcar sintáctico, es decir que pueden ser construidas utilizando una o más construcciones de la sintaxis abstracta. Una notación conveniente para acceder a los campos de una tupla apuntada por un puntero es la flecha ($\rightarrow$). De esta forma, en lugar de acceder a la memoria referenciada por un puntero con la estrella ($\star$), y luego consultar uno de los campos de la tupla con el punto (.), uno puede hacer uso de esta abreviatura que resulta más conveniente.

$$v \rightarrow fn \quad \stackrel{def}{=} \quad \star v.fn$$

Otra notación que resulta útil en el lenguaje, es la de agrupar argumentos. Dada la definición de una función o un procedimiento, puede ocurrir que varios de sus parámetros posean el mismo tipo. En estas ocasiones es conveniente agrupar todas sus declaraciones en una sola. De esta forma queda más claro que todos los argumentos reunidos poseen el mismo tipo.

$$\textbf{fun } f \ (\ \ldots\ a_1, a_2, \ldots, a_n : \theta\ \ldots\ )\ \ldots \quad \stackrel{def}{=} \quad \textbf{fun } f \ (\ \ldots\ a_1 : \theta, a_2 : \theta, \ldots, a_n : \theta\ \ldots\ )\ \ldots$$

Las últimas construcciones para escribir código de manera cómoda en el lenguaje son las variantes de la sentencia *if*. La primer notación, permite omitir el último bloque de sentencias (**else**). La segunda notación, nos da la capacidad de agregar una cantidad arbitraria de condiciones adicionales (**elif**).

$$\textbf{if } b \textbf{ then } ss \quad \stackrel{def}{=} \quad \textbf{if } b \textbf{ then } ss \textbf{ else skip}$$

$$\textbf{if } b_1 \textbf{ then } ss_1 \textbf{ elif } b_2 \textbf{ then } ss_2 \textbf{ else } ss_3 \quad \stackrel{def}{=} \quad \textbf{if } b_1 \textbf{ then } ss_1 \textbf{ else if } b_2 \textbf{ then } ss_2 \textbf{ else } ss_3$$

3.3. Implementación del Parser

Habiendo presentado de manera informal la sintaxis concreta y definido de manera formal la sintaxis abstracta, estamos en condiciones para describir los detalles principales sobre el desarrollo del parser para ΔΔLang. Haremos mención de las decisiones más relevantes tomadas durante su implementación, las dificultades encontradas en el camino, y algunas limitaciones que debieron ser resueltas.

3.3.1. Librerías

Inicialmente se comenzó utilizando la librería *Parsec* [2]. La decisión se tomó debido que algunos de nosotros ya estábamos familiarizados con su uso por proyectos anteriores. Más adelante, en las etapas finales del desarrollo del parser, se decidió migrar el código a *Megaparsec* [3]. La transición fue justificada por las limitaciones que presentaba la primera opción frente a la segunda, que además de solucionar algunas de las dificultades de la implementación de forma sencilla, ofrece un rango de funcionalidades más diverso que puede beneficiar al desarrollo futuro del intérprete.

Parsec

Parsec es una librería para el diseño de parsers monádicos, implementada en *Haskell*, y escrita por *Daan Leijen*. Es simple, segura, rápida, y posee buena documentación. Para la etapa inicial de desarrollo, resultó ser una herramienta intuitiva y fácil de manejar. A pesar de sus cualidades, para este proyecto en particular, la librería no ofrecía la suficiente flexibilidad y funcionalidad que buscábamos. Una primera versión completa del parser fue desarrollada utilizando esta librería.

Megaparsec

Megaparsec se puede considerar el sucesor extraoficial de *Parsec*, escrita por *Mark Karpov*. Partiendo de las bases definidas por esta última, la librería busca ofrecer mayor flexibilidad para la configuración del parser y una generación de mensajes de error más sofisticada respecto a su antecesor. La herramienta resulta familiar para todo el que tenga ciertos conocimientos básicos sobre *Parsec*. La transición de librerías se vio aliviada por esta característica, debido a la semejanza entre ambas.

Breve Comparación

Como mencionamos previamente, debido que *Parsec* no resultó ser la herramienta ideal para el desarrollo de nuestro parser, se decidió comenzar a utilizar *Megaparsec*. Las razones puntuales que nos llevaron a tomar esta decisión se listan a continuación.

- Análisis Léxico: Ambas librerías implementan un mecanismo sencillo para definir un *analizador léxico*, dentro del mismo parser. De esta forma, se simplifica el diseño del parser al unificar estas dos fases fuertemente acopladas. La diferencia entre ambas, radica en el hecho que *Parsec* es demasiado inflexible en este aspecto. Para ciertas cuestiones, se tuvo que redefinir gran parte de la implementación de la librería para poder acomodarla a nuestras necesidades. En cambio, *Megaparsec* no impone ninguna estructura sobre el *analizador léxico*, y solo provee funcionalidades básicas elementales para su definición.

- Mensajes de Error: La generación de mensajes de error en el análisis sintáctico (al igual que para los chequeos estáticos) es una tarea sumamente importante. La primera herramienta utilizada, ofrecía una forma simple y concisa para el informe de errores. En la misma, se podían especificar cuales eran los *tokens* esperados (o inesperados) por el parser al momento de fallar, junto con el mensaje informativo asociado a esta. A pesar de esto, la segunda opción presenta una generación de errores mucho más desarrollada. Además de conservar las funcionalidades previas, se pueden configurar nuevas clases de errores junto con la forma que los mensajes de error son presentados.

- Desarrollo Futuro: Una vez que el desarrollo del parser se encuentre lo suficientemente avanzado, será necesario volver a esta etapa y adecuarla a las nuevas necesidades que

hayan surgido en el camino. En este aspecto, *Megaparsec* ofrece una serie de funcionalidades adicionales que no se encuentran en *Parsec*. Una de ellas es el soporte para múltiples errores, junto con la capacidad de atrapar errores, lo que permite un control mucho más amplio sobre la información que recibe el usuario al analizar su código; las cuales se logran gracias a la recuperación de errores de parseo. También existe la posibilidad de agregar casos de test para aumentar la certeza que el funcionamiento del parser implementado es correcto. Una última característica que puede resultar útil en un futuro, es el parseo *sensible a la indentación* que ofrece la librería.

3.3.2. Información de Posición

Una de las tareas importantes que debe realizar el parser es la generación de mensajes de error informativos y precisos con el fin de facilitar su corrección. Es fundamental entonces poder indicar exactamente *donde* ocurre el error en el programa. Cuando se produce un error durante la etapa de parseo, la librería *Megaparsec* ya incorpora un buen mecanismo para señalar la posición en dónde se ha encontrado la falla. De todas maneras la implementación de los chequeos estáticos del lenguaje, que presentaremos en el siguiente capítulo, también debe poder informar de forma precisa los errores producidos ahora en esta etapa. Para solucionar esto, se implementó la sintaxis abstracta utilizando un método similar al empleado por el compilador de *Haskell* [4], `GHC` [5]. Informalmente la estrategia es encapsular a toda construcción sintáctica que consideremos relevante para la generación de errores con la información de su posición correspondiente en el programa declarado.

El módulo *Syntax.Located* define al *tipo de datos* `Located` que almacena las lineas y columnas de inicio y fin de un determinado elemento, junto con el correspondiente elemento; actualmente el lenguaje no posee un sistema de módulos por lo que la información de lineas y columnas es suficiente, ya que nunca se parsea ni chequea estáticamente más de un archivo.

```
1  -- Parsing Information
2  data Located e = L { info :: Info
3                     , item :: e
4                     }
5
6  -- Position Information
7  data Info = I { sLin :: Line
8                , sCol :: Column
9                , eLin :: Line
10               , eCol :: Column
11               }
```

Código 3.1: Información de Posición en Parser

A medida que avanza el parser, inmediatamente después de parsear algún elemento, este se encapsula con la información de su posición y se almacena en el *árbol de sintaxis abstracta*. Por ejemplo en la implementación de *Syntax.Expr* para las expresiones del lenguaje, podemos almacenar tanto la posición de una expresión particular como todas las de sus subexpresiones. Lo cual tiene como ventaja que en el caso de encontrarse una falla (por ejemplo, un error de tipos) se pueda exhibir toda la *traza* de parseo junto con sus posiciones correspondientes. Notar que en la definición del tipo `Expr` no existen subexpresiones con este mismo tipo, sino que en su lugar tienen tipo `LExpr`, es decir expresiones encapsuladas con la información sobre su posición. Por lo general todo nombre de tipo prefijado con `L` significará que este está encapsulado.

```haskell
-- Expressions
data Expr = Const LConstant          -- Constants
          | Loc LLocation            -- Variables
          | UOp UnOp LExpr           -- Unary Operators
          | BOp BinOp LExpr LExpr    -- Binary Operators
          | FCall Id [LExpr]         -- Function Calls

type LExpr = Located Expr
```

Código 3.2: Expresiones del Lenguaje

3.3.3. Módulos

A continuación describiremos brevemente los distintos módulos en los que se ha dividido la implementación del parser. Haremos mención de los detalles más relevantes de cada uno de estos elementos, junto con el propósito de los mismos.

En *Parser.Position* se proveen todas las funcionalidades necesarias para poder calcular la ubicación en el archivo del elemento que se intenta parsear. Como mencionamos previamente, esta tarea es fundamental para luego poder dar mensajes de errores precisos e informativos al usuario. Haciendo uso de la función `getSourcePos`, provista por la librería, podemos extraer la posición actual del parser y luego, ligarla con el elemento sintáctico correspondiente.

El módulo *Parser.Lexer* es uno de los más importantes, ya que implementa la totalidad del *analizador léxico* del lenguaje. En este se implementan funciones para parsear todas las clases de identificadores, los valores constantes (como los numéricos, por ejemplo), y las *palabras claves* y operadores del lenguaje. Inicialmente cuando se utilizaba la librería *Parsec*, se tuvo que redefinir la mayoría de las funciones que implementaba debido que las mismas consumían automáticamente todos los *whitespaces* (espacios en blanco, comentarios, saltos de línea, etc.) al parsear un elemento. Lo cual impedía poder calcular de manera precisa la posición de las distintas estructuras sintácticas del lenguaje. Luego de la transición, debido que *Megaparsec* delega la responsabilidad del consumo de *whitespace* al usuario, se pudo hacer uso de las funciones auxiliares que brinda la librería, y se simplificó el módulo.

En *Parser.Expr* se parsean las diversas expresiones del lenguaje. Una particularidad interesante de este módulo, es el uso de la función `makeExprParser`. Dado un parser de términos, que serían los elementos básicos que conforman una expresión, y una tabla de operadores, donde se debe especificar la asociatividad y precedencia de cada uno, la función construye un parser para expresiones basado en los mismos. En nuestro caso, se hizo uso de este mecanismo para las expresiones y las variables del lenguaje. Para el primero, su implementación es directa debido que es una situación usual. En cambio, para el segundo, se tuvo que interpretar a las distintas operaciones para el acceso de variables como operadores de expresiones para poder aprovechar la función especificada en la librería.

El módulo *Parser.Statement* se encarga de obtener las sentencias especificadas en el código. A diferencia de la *sintaxis abstracta*, en la implementación del lenguaje se permiten múltiples formas para detallar la instrucción condicional *if*. Para la misma, el componente *else* es opcional, y además, se pueden agregar una cantidad arbitraria de condicionales *elif*. Otra particularidad, es la forma de obtener la ubicación para la asignación. Debido que esta es la única sentencia que no posee un delimitador final, calcular su posición no es una tarea inmediata.

En *Parser.Decl* se parsean todas las declaraciones del lenguaje. Por un lado se tienen a las definiciones de tipo, las cuales abarcan a las enumeraciones, los sinónimos, y las tuplas. Por el otro tenemos a las definiciones de funciones y procedimientos, junto con todos los elementos que

las conforman, como sus parámetros y restricciones.

Los módulos restantes no presentan complejidad adicional en comparación con los descriptos recientemente. El módulo *Parser.Type* obtiene los distintos tipos sintácticamente válidos del lenguaje. En *Parser.Class* se parsean todas las clases predefinidas en el lenguaje. Y finalmente, el parser para programas se implementa en *Parser.Program*.

Capítulo 4

Sobre los chequeos estáticos

En este capítulo, definiremos formalmente los distintos chequeos estáticos del lenguaje y presentaremos algunos aspectos interesantes sobre su primera implementación. Los fundamentos teóricos utilizados en esta sección están basados en *Theories of Programming Languages* de *Reynolds* [6]. Los capítulos sobre el sistema de tipos (`15`), el subtipado (`16`), y el polimorfismo (`18`) resultaron indispensables para el desarrollo del informe. Por el otro lado, la implementación de los chequeos estáticos siguió la idea planteada en el artículo de *Castegren* y *Reyes* [7]. Las secciones más relevantes para la realización de nuestro trabajo comprenden la definición del *typechecker*, el soporte para *backtraces*, y el agregado de *warnings*.

4.1. Metavariables

A lo largo del capítulo, se utilizarán diversas metavariables (a veces acompañadas de supraíndices, o subíndices) para representar distintas clases de construcciones sintácticas. A continuación se listará cada una junto con el elemento sintáctico que comúnmente simbolizará, a menos que se indique lo contrario en el momento oportuno.

Expresiones			
e	$\langle expression \rangle$	ct	$\langle constant \rangle$
v	$\langle variable \rangle$	n	$\langle integer \rangle$
x, a	$\langle id \rangle$	r	$\langle real \rangle$
$\oplus$	$\langle binary \rangle$	b	$\langle bool \rangle$
$\ominus$	$\langle unary \rangle$	c	$\langle character \rangle$
Sentencias			
s	$\langle sentence \rangle$	ss	$\langle sentences \rangle$
Tipos			
θ	$\langle type \rangle$	as	$\langle arraysize \rangle$
td	$\langle typedecl \rangle$	tn	$\langle tname \rangle$
tv	$\langle typevariable \rangle$	cn	$\langle cname \rangle$
cl	$\langle class \rangle$	fn	$\langle fname \rangle$
io	$\langle io \rangle$	fd	$\langle field \rangle$
Programas			
fpd	$\langle funprocdecl \rangle$	fa	$\langle funargument \rangle$
vd	$\langle variabledecl \rangle$	fr	$\langle funreturn \rangle$
cs	$\langle constraints \rangle$	pa	$\langle procargument \rangle$

4.2. Notación

A continuación presentaremos algo de notación general sobre los diferentes *juicios de tipado* que nos encontraremos en este capítulo. Por lo general, diremos que un elemento sintáctico está *bien tipado* cuando podamos construir una derivación de su respectivo juicio. Comúnmente, utilizaremos la siguiente notación para decir que la construcción sintáctica χ está *bien tipada* bajo el contexto π, en base a las reglas de la categoría sintáctica γ.

$$\pi \vdash_\gamma \chi$$

En algunas ocasiones estos juicios prueban que la extensión de algún contexto determinado es válida. De alguna manera podemos pensar que estos juicios tienen a tal extensión como resultado de su chequeo. En estas situaciones se utilizará la siguiente notación, donde ω representa la extensión de alguno de los contextos involucrados.

$$\pi \vdash_\gamma \chi : \omega$$

Para facilitar la lectura muchas veces agrupamos (o por el contrario, desagrupamos) diferentes tipos de contextos. Debido que comúnmente se deberá almacenar información de distintos

índoles para efectuar la verificación, se hará uso de la siguiente notación para reflejar esta condición. Notar que también existe la posibilidad que no se necesite utilizar ninguna información contextual, por lo que en estos escenarios se omite el listado de contextos.

$$\pi_1, \ldots, \pi_n \vdash_\gamma \chi : \omega$$

Ciertas construcciones sintácticas podrán ser consideradas *bien tipadas* de acuerdo a determinados contextos locales $\overline{\pi}_j$, en estas situaciones se empleará la siguiente notación. Los contextos mencionados representan información elemental necesaria para la derivación del juicio de tipado apropiado.

$$\pi_1, \ldots, \pi_n \overset{\overline{\pi}_1,\ldots,\overline{\pi}_m}{\vdash_\gamma} \chi : \omega$$

A lo largo de este capítulo, se darán distintos conjuntos de reglas γ en base al elemento sintáctico que χ represente. Al mismo tiempo, se definirán diversos contextos π que almacenarán la información recopilada a lo largo de una, o más, derivaciones de distintos juicios.

Los contextos que emplearemos a lo largo del capítulo serán conjuntos compuestos por *n-uplas* de elementos sintácticos del lenguaje. Debido que la operación principal sobre estas construcciones será el agregado de distintas componentes que conforman al elemento sintáctico recientemente analizado, se hará uso de la siguiente abreviatura para simplificar la notación.

$$(\chi_1, \ldots, \chi_n) \triangleright \pi \overset{def}{=} \{(\chi_1, \ldots, \chi_n)\} \cup \pi$$

4.3. Contextos y Juicios

Para asegurar la corrección estática de un programa, se deben validar cada una de sus componentes, y en el caso que todas superen su respectiva verificación, se dirá que el mismo se encuentra *bien tipado*. Según la sintaxis del lenguaje, un programa posee la siguiente estructura, donde vale que $n \geq 0$ y $m > 0$; lo cual implica que un programa tiene al menos una función o un procedimiento declarado.

$$\begin{gathered} typedecl_1 \\ \ldots \\ typedecl_n \\ funprocdecl_1 \\ \ldots \\ funprocdecl_m \end{gathered}$$

4.3.1. Chequeos para Tipos

Las primeras reglas que definiremos serán sobre los tipos que ocurren en el programa. Con las cuales se buscan verificar propiedades tales como el uso adecuado de los tipos definidos por el usuario, y limitar la ocurrencia de variables de tipo e identificadores para la abstracción de tamaños fijos de arreglos. Comenzamos declarando los distintos contextos que serán necesarios para la definición de los juicios y las reglas.

Contextos para Declaración de Tipos

Una definición de tipo *typedecl* consiste en alguna de las siguientes tres construcciones sintácticas; un tipo enumerado, un sinónimo de tipo, o una estructura de tipo tupla. Cuando una de estas declaraciones se encuentre *bien tipada* su información será almacenada en el contexto adecuado. Habrá un contexto diferente para cada una de las categorías de tipos que se pueden definir en el lenguaje. Además, estos conjuntos tendrán una serie de invariantes que definirán cuando un contexto está correctamente construido.

- **enum** $tn = cn_1, cn_2, \ldots, cn_m$
- **syn** tn **of** $tv_1, \ldots, tv_l = \theta$
- **tuple** tn **of** $tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m$

En el caso de la declaración de un tipo enumerado, se debe almacenar el nombre del tipo definido junto con el listado de constantes enumeradas en su cuerpo. Una invariante que se debe respetar en este contexto, es que los nombres de constantes deben ser únicos. Esto significa que no pueden ser repetidos dentro de una misma definición, ni tampoco ocurrir en otras.

$$\pi_e = \{(tn, \{cn_1, \ldots, cn_m\}) \mid tn \in \langle tname \rangle \wedge cn_i \in \langle cname \rangle\}$$

Para los sinónimos, además del nombre, se deben guardar las variables de tipo utilizadas como argumentos, junto con el tipo que lo define. En este contexto, la invariante debe asegurar que dentro de una declaración no se repitan los identificadores empleados para representar a sus parámetros de tipo.

$$\pi_s = \{(tn, \{tv_1, \ldots, tv_l\}, \theta) \mid tn \in \langle tname \rangle \wedge tv_i \in \langle typevariable \rangle \wedge \theta \in \langle type \rangle\}$$

Finalmente, para las tuplas, tenemos que almacenar su nombre, sus argumentos de tipo, y los distintos campos especificados en su definición. En esta situación, además de evitar la repetición de variables de tipo, se tiene que asegurar que los identificadores de campo sean únicos dentro del cuerpo de la declaración.

$$\pi_t = \{(tn, \{tv_1, \ldots, tv_l\}, \{fd_1, \ldots, fd_m\}) \mid tn \in \langle tname \rangle \wedge tv_i \in \langle typevariable \rangle \wedge fd_j \in \langle field \rangle\}$$

Estos tres contextos se encargarán de almacenar toda la información relacionada con los tipos declarados. A todas las condiciones de consistencia mencionadas anteriormente se le tiene que sumar una última, los nombres de tipos definidos deben ser únicos. Es decir, que no puede haber más de una definición para el mismo identificador de tipo entre los distintos contextos.

Contexto para Variables de Tipo

En general, la definición de cualquier regla de tipado para un tipo deberá tener un listado de las variables de tipo presentes en el alcance actual. Ya sea como parámetro de tipo en una declaración de tipo, o como tipo polimórfico para algún parámetro de una función o un procedimiento, solo se deben permitir el uso de estas variables cuando se han introducido previamente en el código.

$$\pi_{tv} \subseteq \langle typevariable \rangle$$

Contexto para Identificadores de Tamaños

Similar como sucede con las variables de tipo, hay una necesidad de llevar registro de todos los identificadores introducidos para abstraer los tamaños de arreglos. En el prototipo de funciones y procedimientos, es posible especificar identificadores para las dimensiones de arreglos. Lo cual permite que dentro de sus cuerpos, puedan ser utilizados en la declaración de nuevos arreglos cuyas dimensiones coincidirán con los arreglos correspondientes en el encabezado.

$$\pi_{sn} \subseteq \langle sname \rangle$$

Reglas para Tipos

En un programa, hay tres situaciones diferentes donde se puede escribir un tipo, y en base a esta, las propiedades que se deben verificar varían levemente. Un tipo puede ocurrir en una declaración de tipo, en una definición del prototipo de una función o un procedimiento, e incluso en la declaración de una variable dentro del cuerpo de una función o un procedimiento.

Cuando nos encontramos analizando un tipo, utilizaremos la siguiente notación para denotar que el tipo θ es válido en el contexto de los tipos definidos; enumerados π_e, sinónimos π_s, y tuplas π_t. Sumados a los mismos, los conjuntos π_{tv} de variables de tipo determinará cuando la escritura de un tipo polimórfico particular está permitida, y tenemos además π_{sn} para los identificadores de tamaños de arreglos. Utilizando una notación más compacta, comúnmente haremos referencia al contexto $\pi_{\mathbf{T}}$ para representar a la anterior tripla de contextos. Mientras que emplearemos $\overline{\pi}$ para simbolizar a los últimos dos conjuntos. Esta salvedad la tendremos para facilitar la lectura de las reglas, y poder concentrarnos propiamente en las derivaciones.

$$\pi_e, \pi_s, \pi_t \overset{\pi_{tv}, \pi_{sn}}{\vdash_t} \theta$$

$$\pi_{\mathbf{T}} \overset{\overline{\pi}}{\vdash_t} \theta$$

Para decidir si uno de estos *juicios* es válido, tenemos que proveer una derivación o prueba, utilizando las reglas que definiremos a continuación.

Comenzaremos con los tipos básicos del lenguaje. La prueba de los mismos es inmediata, ya que su regla no presenta ninguna premisa. Por lo tanto, todo tipo básico es considerado un tipo correcto.

Regla para Tipos 1. Básicos

$$\frac{}{\pi_{\mathbf{T}} \overset{\overline{\pi}}{\vdash_t} \theta} \quad \text{cuando } \theta \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{char}\}$$

Un puntero será correcto, siempre que el tipo del valor al que hace referencia sea correcto. Notar que la premisa de la regla requiere de la prueba de un tipo estructuralmente menor al inicial.

Regla para Tipos 2. Punteros

$$\frac{\pi_{\mathbf{T}} \overset{\overline{\pi}}{\vdash_t} \theta}{\pi_{\mathbf{T}} \overset{\overline{\pi}}{\vdash_t} \mathbf{pointer}\ \theta}$$

Para los arreglos, se necesitarán verificar los tamaños de sus dimensiones, junto con el tipo de valores que almacenará. Notar que para analizar los primeros, solo se requiere la información de los identificadores de tamaños presentes en el contexto local.

Regla para Tipos 3. Arreglos

$$\frac{\pi_{\mathbf{T}} \overset{\pi_{tv},\pi_{sn}}{\vdash_t} \theta \qquad \overset{\pi_{sn}}{\vdash_{as}} as_i}{\pi_{\mathbf{T}} \overset{\pi_{tv},\pi_{sn}}{\vdash_t} \textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ \theta}$$

Para las dimensiones de un arreglo, se pueden especificar sus tamaños de dos maneras. Si el mismo es un valor natural, la verificación es inmediata ya que cualquier número natural representa un tamaño de arreglo válido. En cambio, cuando se utilizan identificadores de tamaños, es necesario comprobar que hayan sido previamente introducidos, donde el único lugar de un programa donde se permiten introducir esta clase de identificadores es en el prototipo de funciones y procedimientos.

Regla para Tipos 4. Tamaños Concretos

$$\frac{}{\overset{\pi_{sn}}{\vdash_{as}} as} \quad \text{cuando } as \in \langle natural \rangle$$

Regla para Tipos 5. Identificadores de Tamaños

$$\frac{as \in \pi_{sn}}{\overset{\pi_{sn}}{\vdash_{as}} as}$$

Al utilizar una variable de tipo, es necesario verificar que su uso sea válido en el alcance actual. Similar a los identificadores de tamaños para arreglos, se permiten introducir esta clase de variables en el encabezado de funciones y procedimientos, lo que luego posibilitará emplearlas en sus respectivos cuerpos. Adicionalmente existe la posibilidad de declarar un tipo de forma paramétrica, lo que concede la capacidad de usar las variables de tipo especificadas como argumento, en la definición del nuevo tipo.

Regla para Tipos 6. Variables de Tipo

$$\frac{tv \in \pi_{tv}}{\pi_{\mathbf{T}} \overset{\pi_{tv},\pi_{sn}}{\vdash_t} tv}$$

Las reglas para los tipos definidos, pueden separarse en dos categorías de acuerdo si los mismos poseen argumentos de tipo o no. La verificación de un tipo definido no parametrizado, consiste simplemente de constatar que su nombre se encuentra declarado en alguno de los contextos de tipos correspondientes. Evidentemente, hay que asegurar que en su definición no se haya especificado ningún argumento de tipo.

Regla para Tipos 7. Tipos Enumerados

$$\frac{(tn, \{cn_1, \ldots, cn_m\}) \in \pi_e}{\pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} tn}$$

Regla para Tipos 8. Sinónimos sin Argumentos

$$\frac{(tn, \emptyset, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} tn}$$

Regla para Tipos 9. Tuplas sin Argumentos

$$\frac{(tn, \emptyset, \{fd_1, \ldots, fd_m\}) \in \pi_t}{\pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} tn}$$

En cambio, para un tipo parametrizado, es necesario realizar unas verificaciones adicionales. En particular, se deben validar todos los tipos especificados como argumentos del mismo, y asegurar que la cantidad de argumentos coincida con el número de parámetros declarados en la definición del tipo.

Regla para Tipos 10. Sinónimos con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_l\}, \theta) \in \pi_s \qquad \pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} \theta_i}{\pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} tn \textbf{ of } \theta_1, \ldots, \theta_l}$$

Regla para Tipos 11. Tuplas con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_l\}, \{fd_1, \ldots, fd_m\}) \in \pi_t \qquad \pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} \theta_i}{\pi_e, \pi_s, \pi_t \overset{\overline{\pi}}{\vdash_t} tn \textbf{ of } \theta_1, \ldots, \theta_l}$$

Ejemplo de Prueba

Supongamos los siguientes contextos donde no hay declarados tipos enumerados, ni sinónimos de tipo, y solo tenemos definido el tipo tupla *node* parametrizado en *Z*. Notar que el campo *elem* es de tipo variable *Z*, y *next* es un puntero al mismo tipo.

$$\pi_e = \emptyset$$
$$\pi_s = \emptyset$$
$$\pi_t = \{(node, \{Z\}, \{elem : Z, next : \textbf{pointer } node \textbf{ of } Z\})\}$$

Sumados a los contextos anteriores, también se encuentran definidos los siguientes conjuntos. No hay ningún identificador de tamaño declarado en el alcance actual, y solo se ha introducido la variable de tipo *A*.

$$\pi_{sn} = \emptyset$$
$$\pi_{tv} = \{A\}$$

Si nos adelantamos un poco, y quisiéramos probar la declaración del sinónimo de tipo **syn** *list* **of** A = **pointer** *node* **of** A, entonces una parte de la prueba consistirá en demostrar la validez del tipo que ocurre dentro de la definición. Por lo tanto, con los contextos previos, se puede realizar la siguiente derivación. Notar el uso de las reglas para punteros, tuplas definidas, y variables de tipo.

Prueba 1. Demostración de corrección para el tipo *puntero a nodo*.

$$\dfrac{\dfrac{(node, \{Z\}, \{elem : Z, next : \textbf{pointer } node \textbf{ of } Z\}) \in \pi_t \qquad \dfrac{A \in \pi_{tv}}{\pi_e, \pi_s, \pi_t \overset{\pi_{tv}, \pi_{sn}}{\vdash_t} A}\,(6)}{\pi_e, \pi_s, \pi_t \overset{\pi_{tv}, \pi_{sn}}{\vdash_t} node \textbf{ of } A}\,(11)}{\pi_e, \pi_s, \pi_t \overset{\pi_{tv}, \pi_{sn}}{\vdash_t} \textbf{pointer } node \textbf{ of } A}\,(2)$$

4.3.2. Variables de Tipo Libres

En una declaración de tipo, las únicas variables que se pueden y deben utilizar son las que fueron previamente introducidas como parámetros de la declaración. Es importante remarcar que no existe la posibilidad de introducir una variable de tipo que luego no ocurre en la definición del tipo. Lo cual es una diferencia con los prototipos de funciones y procedimientos en donde se permite introducir variables de tipo que luego pueden no ocurrir nunca en su respectivo cuerpo.

$$FTV : \langle type \rangle \to \{\ \langle typevariable \rangle\ \}$$

Debido a todo esto, necesitamos definir cuando una variable de tipo es considerada *libre*. En el lenguaje no hay ninguna clase de cuantificación, a nivel de tipos, para estos elementos; pero nos referiremos de esta manera informal a todas las variables que ocurran dentro de uno. El propósito de esta definición es simplificar la escritura de algunas reglas.

$$\begin{aligned}
&FTV(\textbf{int}) && = \emptyset \\
&FTV(\textbf{real}) && = \emptyset \\
&FTV(\textbf{bool}) && = \emptyset \\
&FTV(\textbf{char}) && = \emptyset \\
&FTV(\textbf{pointer}\ \theta) && = FTV(\theta) \\
&FTV(\textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ \theta) && = FTV(\theta) \\
&FTV(tv) && = \{tv\} \\
&FTV(tn) && = \emptyset \\
&FTV(tn\ \textbf{of}\ \theta_1, \ldots, \theta_n) && = FTV(\theta_1) \cup \ldots \cup FTV(\theta_n)
\end{aligned}$$

4.3.3. Chequeos para Declaración de Tipos

En el chequeo de declaraciones de tipo se debe garantizar la unicidad de los identificadores empleados para representar determinadas componentes, como los nombres de constantes enumeradas, los campos de tupla, y los parámetros de tipo. Adicionalmente debemos probar la validez de los tipos especificados dentro de las declaraciones. Incluso, hay que asegurar el uso adecuado de las variables de tipo introducidas como parámetros de las definiciones.

Reglas para Declaración de Tipos

La prueba de una serie de declaraciones de tipo responde al orden en que las mismas se encuentran especificadas, lo cual implica que las reglas no permiten la definición mutua entre estas declaraciones. De esta manera, un tipo definido solo será accesible para las declaraciones posteriores al mismo.

El *juicio de tipado* que prueba la validez de una declaración de tipo será aquel que tiene una derivación que prueba la correcta extensión del contexto original con la nueva declaración. Recordar que al realizarse estas extensiones, se deben seguir respetando las invariantes de consistencia para los conjuntos involucrados.

$$\pi_{\mathbf{T}} \vdash_{td} typedecl : \pi'_{\mathbf{T}}$$

La regla para la definición de tipos enumerados es simple, debido a las invariantes de los contextos de tipos. Notar que una extensión válida del contexto original ya nos asegura la unicidad del nombre de tipo respecto a las otras definiciones, y que los constructores empleados en la declaración no son utilizados en otras definiciones de tipos enumerados.

Regla para Declaración de Tipos 12. Enumerados

$$\frac{}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{enum}\ tn = cn_1, \ldots, cn_m : \pi'_e, \pi_s, \pi_t}$$

donde $\pi'_e = (tn, \{cn_1, \ldots, cn_m\}) \triangleright \pi_e$.

Para los sinónimos se tiene que asegurar que el conjunto de parámetros de la declaración coincida con el conjunto de variables de tipo utilizadas en su definición. Esta condición evita la ocurrencia de variables *libres* en el cuerpo de la declaración, y también obliga el uso de todos los argumentos de la misma. Además el tipo que propiamente define al sinónimo tiene que ser válido. Notar que no se permiten la utilización de identificadores para tamaños de arreglos.

Regla para Declaración de Tipos 13. Sinónimos sin Argumentos

$$\frac{\pi_e, \pi_s, \pi_t \overset{\emptyset_{tv}, \emptyset_{sn}}{\vdash_t} \theta}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{syn}\ tn = \theta : \pi_e, \pi'_s, \pi_t}$$

donde $\pi'_s = (tn, \emptyset, \theta) \triangleright \pi_s$.

Regla para Declaración de Tipos 14. Sinónimos con Argumentos

$$\frac{\pi_{tv} \subseteq FTV(\theta) \qquad \pi_e, \pi_s, \pi_t \overset{\pi_{tv}, \emptyset_{sn}}{\vdash_t} \theta}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{syn}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = \theta : \pi_e, \pi'_s, \pi_t}$$

donde $\pi'_s = (tn, \{tv_1, \ldots, tv_l\}, \theta) \triangleright \pi_s$, y $\pi_{tv} = \{tv_1, \ldots, tv_l\}$.

Las verificaciones para tuplas son similares a las de sinónimos, salvo que se deben adecuar para los múltiples campos de la misma. Hay que asegurar la igualdad entre los parámetros de la definición, y las variables de tipo que ocurren en todos los campos. Adicionalmente, se tienen que analizar todos los tipos para asegurar su corrección. Por último, se deben respetar las invariantes de unicidad tanto para los nombres de campos, como para los argumentos de tipo.

Regla para Declaración de Tipos 15. Tuplas sin Argumentos

$$\frac{\pi_e, \pi_s, \pi_t \overset{\emptyset_{tv}, \emptyset_{sn}}{\vdash_t} \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \emptyset, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$.

Regla para Declaración de Tipos 16. Tuplas con Argumentos

$$\frac{\pi_{tv} \subseteq FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) \qquad \pi_e, \pi_s, \pi_t \overset{\pi_{tv}, \emptyset_{sn}}{\vdash_t} \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \{tv_1, \ldots, tv_l\}, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$, y $\pi_{tv} = \{tv_1, \ldots, tv_l\}$.

Las siguientes reglas son las que permiten la definición de tipos recursivos, donde hay que notar lo restrictivas que resultan. La única posibilidad de declarar un tipo que se define en términos de si mismo es mediante el uso de punteros dentro de tuplas. Adicionalmente solo se permiten utilizar las mismas variables de tipo paramétricas que en la definición, e incluso se las debe especificar en el mismo orden.

Regla para Declaración de Tipos 17. Recursión para Tuplas sin Argumentos

$$\frac{\theta_i \neq \textbf{pointer}\ tn \implies \pi_e, \pi_s, \pi_t \overset{\emptyset_{tv},\emptyset_{sn}}{\vdash_t} \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \emptyset, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$.

Regla para Declaración de Tipos 18. Recursión para Tuplas con Argumentos

$$\frac{\pi_{tv} \subseteq FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) \qquad \theta_i \neq \textbf{pointer}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l \implies \pi_e, \pi_s, \pi_t \overset{\pi_{tv},\emptyset_{sn}}{\vdash_t} \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

donde $\pi'_t = (tn, \{tv_1, \ldots, tv_l\}, \{fn_1 : \theta_1, \ldots, fn_m : \theta_m\}) \triangleright \pi_t$, y $\pi_{tv} = \{tv_1, \ldots, tv_l\}$.

Ejemplo de Prueba

Siguiendo con el ejemplo especificado previamente, presentaremos la prueba de corrección para dos declaraciones de tipo particulares. Primero verificaremos una estructura de tipo tupla, y luego realizaremos lo apropiado para un sinónimo de tipo. Supongamos los siguientes contextos, los cuales serán los resultados obtenidos luego de realizados los respectivos análisis.

$$\pi_s = \{(list, \{A\}, \textbf{pointer}\ node\ \textbf{of}\ A)\}$$
$$\pi_t = \{(node, \{Z\}, \{elem : Z, next : \textbf{pointer}\ node\ \textbf{of}\ Z\})\}$$

Comenzando con los contextos de tipos vacíos, se puede realizar la prueba para la estructura de tipo tupla *node*. Como se preciso anteriormente, debido que el campo *next* realiza una llamada recursiva al tipo que estamos definiendo, se omite la verificación de su tipo. Notar el uso de las reglas para las variables de tipo, y la recursión en declaración de tuplas.

Prueba 2. Derivación de corrección para la declaración de tipo *node*.

$$\frac{\{Z\} \subseteq FTV(Z) \cup FTV(\textbf{pointer}\ node\ \textbf{of}\ Z) \qquad \dfrac{Z \in \{Z\}_{tv}}{\emptyset_e, \emptyset_s, \emptyset_t \overset{\{Z\}_{tv},\emptyset_{sn}}{\vdash_t} Z}\ (6)}{\emptyset_e, \emptyset_s, \emptyset_t \vdash_{td} \textbf{tuple}\ node\ \textbf{of}\ Z = elem : Z, next : \textbf{pointer}\ node\ \textbf{of}\ Z : \emptyset_e, \emptyset_s, \pi_t}\ (18)$$

Utilizando el contexto de tuplas obtenido en la derivación anterior, se puede realizar la prueba del sinónimo de tipo *list*. La demostración de corrección del tipo que define al sinónimo ya fue precisada en la sección previa. Por lo tanto, aplicando la regla para sinónimos con parámetros, junto con la demostración mencionada, podemos construir la derivación del *juicio de tipado* correspondiente a la declaración.

Prueba 3. Derivación de corrección para la declaración de tipo *list*.

$$\frac{\{A\} \subseteq FTV(\textbf{pointer}\ node\ \textbf{of}\ A) \qquad \dfrac{\text{Prueba 1}}{\emptyset_e, \emptyset_s, \pi_t \overset{\{A\}_{tv},\emptyset_{sn}}{\vdash_t} \textbf{pointer}\ node\ \textbf{of}\ A}\ (2)}{\emptyset_e, \emptyset_s, \pi_t \vdash_{td} \textbf{syn}\ list\ \textbf{of}\ A = \textbf{pointer}\ node\ \textbf{of}\ A : \emptyset_e, \pi_s, \pi_t}\ (14)$$

Notar que empleando las distintas reglas introducidas a lo largo del capítulo, somos capaces de probar la validez de la declaración para la *lista abstracta* en el lenguaje. El fragmento de código ilustrado (Código 4.1) comprende la implementación, utilizando la sintaxis concreta de ΔΔLang, de las definiciones de tipo analizadas. En las declaraciones se utilizan diferentes variables de tipo con el propósito de dar ejemplos con derivaciones más interesantes.

```
1 type node of (Z) = tuple
2                    elem : Z,
3                    next : pointer of node of (Z)
4                    end tuple
5
6 type list of (A) = pointer of node of (A)
```

Código 4.1: Implementación de Lista Abstracta

4.3.4. Identificadores para Tamaños de Arreglos

Al igual que para las variables de tipo, solo se pueden utilizar identificadores de tamaños en determinados lugares de un programa. El tamaño de un arreglo es siempre constante, pero en el lenguaje existe la posibilidad de trabajar con arreglos abstrayéndonos de sus tamaños concretos, al menos de forma estática.

En el prototipo de una función o un procedimiento, es posible introducir identificadores para tamaños de arreglos cuando se definen los tipos de sus argumentos. Lo cual permite que luego, en el respectivo cuerpo de la función o el procedimiento, se utilicen estos identificadores para declarar nuevos arreglos. Incluso se podrán emplear estos identificadores como valores constantes.

$$DAS : \langle type \rangle \to \{\ \langle sname \rangle\ \}$$

Debido a todo esto, necesitamos definir una función que calcule todos los identificadores para tamaños que ocurren en un tipo particular. De esta manera, podremos listar todos los tamaños variables que son introducidos en el prototipo de la función o el procedimiento.

$$\begin{aligned}
&DAS(\textbf{int}) && = \emptyset \\
&DAS(\textbf{real}) && = \emptyset \\
&DAS(\textbf{bool}) && = \emptyset \\
&DAS(\textbf{char}) && = \emptyset \\
&DAS(\textbf{pointer}\ \theta) && = DAS(\theta) \\
&DAS(\textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ \theta) && = \{as_i \mid as_i \in \langle sname \rangle\} \cup DAS(\theta) \\
&DAS(tv) && = \emptyset \\
&DAS(tn) && = \emptyset \\
&DAS(tn\ \textbf{of}\ \theta_1, \ldots, \theta_n) && = DAS(\theta_1) \cup \ldots \cup DAS(\theta_n)
\end{aligned}$$

4.3.5. Chequeos para Funciones y Procedimientos

Las reglas para chequear funciones y procedimientos deberán verificar propiedades como la unicidad de los nombres utilizados para identificar a sus parámetros, como también analizar la validez de sus respectivos tipos. Además con respecto al chequeo del cuerpo de funciones o procedimientos, adelantamos que será formalizado en la siguiente sección una vez que se haya precisado la información contextual necesaria para efectuarlo, junto con las reglas apropiadas para chequear las declaraciones de variables y sentencias que conforman al cuerpo.

Contextos para Funciones y Procedimientos

Una declaración de función o procedimiento $funprocdecl$, puede tener alguna de las dos siguientes formas en base a cual de las construcciones mencionadas define. Similar a la declaración de tipos, cuando uno de estos elementos se encuentra *bien tipado* su información es almacenada en el contexto adecuado.

- **fun** $f\ (a_1 : \theta_1, \ldots, a_l : \theta_l)$ **ret** $a_r : \theta_r$
  where cs
  in $body$
- **proc** $p\ (io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l)$
  where cs
  in $body$

En el caso de las funciones, el contexto contendrá su identificador f, junto con todos los elementos de su prototipo. Los cuales comprenden a sus argumentos $a_1 : \theta_1, \ldots, a_l : \theta_l$, su retorno $a_r : \theta_r$, y las restricciones para las variables de tipo especificadas en su encabezado cs. Por construcción, el contexto de funciones solo es válido si se cumple que el nombre utilizado para referirse a una función determinada es único. Además los identificadores empleados para sus parámetros no deben repetirse dentro del prototipo de la definición.

$$\pi_f = \{(f, \{fa_1, \ldots, fa_l\}, fr, cs) \mid f \in \langle id \rangle \wedge fa_i \in \langle funargument \rangle \wedge \ldots \\ \ldots \wedge fr \in \langle funreturn \rangle \wedge cs \in \langle constraints \rangle\}$$

El contexto para procedimientos cumple un rol análogo que el empleado para funciones. Debe almacenar al identificador p declarado, junto con sus parámetros $io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l$, y las restricciones cs asociadas al mismo. Las invariantes son idénticas al contexto previo.

$$\pi_p = \{(p, \{pa_1, \ldots, pa_l\}, cs) \mid p \in \langle id \rangle \wedge pa_i \in \langle procargument \rangle \wedge cs \in \langle constraints \rangle\}$$

Reglas para Funciones y Procedimientos

A la hora de chequear una función o un procedimiento, utilizaremos la siguiente notación para denotar que la construcción sintáctica extiende correctamente un determinado contexto, cuando además contamos con los tipos definidos en el programa $\pi_{\mathbf{T}}$. Las funciones válidas extenderán al contexto π_f, mientras que los procedimientos harán lo propio con π_p. Utilizando una abreviatura para la notación, nos referiremos al contexto $\pi_{\mathbf{FP}}$ para representar a los correspondientes conjuntos mencionados.

$$\pi_{\mathbf{T}}, \pi_f, \pi_p \vdash_{fp} funprocdecl : \pi'_f, \pi'_p$$
$$\pi_{\mathbf{T}}, \pi_{\mathbf{FP}} \vdash_{fp} funprocdecl : \pi'_{\mathbf{FP}}$$

La verificación de una declaración de función es compleja, donde es necesario validar todos los tipos de sus respectivos parámetros. Notar que en esta instancia se permite la introducción de variables de tipo y de identificadores para tamaños de arreglos por lo que, permitiendo un abuso de notación, $\langle typevariable \rangle_{tv}$ y $\langle sname \rangle_{sn}$ representarán respectivamente a los conjuntos numerables con todas las variables e identificadores al momento de analizar los argumentos de la función. Con los elementos introducidos en estos se conformarán los contextos π_{tv} y π_{sn} que se utilizarán para, entre otras cosas, chequear el tipo del retorno en el cual no se pueden introducir variables de tipo ni identificadores de tamaño. Asumiremos que las restricciones están siempre

bien tipadas dado que el lenguaje solo tiene definidas algunas clases nativas y no existe un mecanismo para definir nuevas, por lo tanto no se presenta ninguna premisa al respecto. Finalmente con toda la información contextual necesaria, y extendiendo el contexto correspondiente a funciones para permitir llamadas recursivas, se deberá verificar el cuerpo de la función. Notar que para su análisis se indexa al *juicio de tipado* con el identificador de la función.

Regla para Función/Procedimiento 19. Funciones

$$\frac{\pi_{\mathbf{T}} \overset{\langle typevariable\rangle_{tv},\langle sname\rangle_{sn}}{\vdash_t} \theta_i \qquad \pi_{\mathbf{T}} \overset{\pi_{tv},\pi_{sn}}{\vdash_t} \theta_r \qquad \pi_{\mathbf{T}}, \overline{\pi}_f, \pi_p, \pi_{tv}, \pi_{sn} \vdash_b^f body}{\pi_{\mathbf{T}}, \pi_f, \pi_p \vdash_{fp} \mathbf{fun}\ f\ (a_1 : \theta_1, \ldots, a_l : \theta_l)\ \mathbf{ret}\ a_r : \theta_r\ \mathbf{where}\ cs\ \mathbf{in}\ body : \overline{\pi}_f, \pi_p}$$

donde $\pi_{tv} = FTV(\theta_1) \cup \ldots \cup FTV(\theta_l)$, y $\pi_{sn} = DAS(\theta_1) \cup \ldots \cup DAS(\theta_l)$. Adicionalmente, se extiende el contexto de funciones con el prototipo en cuestión, para permitir llamadas recursivas $\overline{\pi}_f = (f, \{a_1 : \theta_1, \ldots, a_l : \theta_l\}, a_r : \theta_r, cs) \triangleright \pi_f$.

La verificación de una declaración de procedimiento, al igual que para las funciones, es compleja. Se deben analizar individualmente todos los tipos de sus respectivos parámetros. De la misma manera que en la regla previa, definiremos contextos con los identificadores para tamaños de arreglos y las variables de tipo que se hayan especificado en estos. Por último una vez obtenida toda la información contextual necesaria, y extendiendo el contexto correspondiente a procedimientos para permitir llamadas recursivas, se deberá verificar el cuerpo del procedimiento. Notar que el *juicio de tipado* es indexado con el identificador del procedimiento.

Regla para Función/Procedimiento 20. Procedimientos

$$\frac{\pi_{\mathbf{T}} \overset{\langle typevariable\rangle_{tv},\langle sname\rangle_{sn}}{\vdash_t} \theta_i \qquad \pi_{\mathbf{T}}, \pi_f, \overline{\pi}_p, \pi_{tv}, \pi_{sn} \vdash_b^p body}{\pi_{\mathbf{T}}, \pi_f, \pi_p \vdash_{fp} \mathbf{proc}\ p\ (io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l)\ \mathbf{where}\ cs\ \mathbf{in}\ body : \pi_f, \overline{\pi}_p}$$

donde $\pi_{tv} = FTV(\theta_1) \cup \ldots \cup FTV(\theta_l)$, y $\pi_{sn} = DAS(\theta_1) \cup \ldots \cup DAS(\theta_l)$. Adicionalmente, se extiende el contexto de procedimientos con el prototipo en cuestión, para permitir llamadas recursivas $\overline{\pi}_p = (p, \{io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l\}, cs) \triangleright \pi_p$.

Ejemplo de Prueba

Tomando como base las derivaciones de ejemplos pasadas, presentaremos la prueba de corrección para una declaración de función y otra para una declaración de procedimiento. La primera se utiliza para decidir si una lista es vacía, mientras que el segundo elimina al primer elemento de la lista. Supongamos que los contextos para funciones y procedimientos son los descriptos a continuación, mientras que los de tipos definidos son idénticos a los especificados previamente.

$$\pi_f = \{(isEmpty, \{l : list\ \mathbf{of}\ T\}, b : bool, \emptyset)\}$$
$$\pi_p = \{(tail, \{\mathbf{in/out}\ l : list\ \mathbf{of}\ T\}, \emptyset)\}$$

Debido que tanto la función como el procedimiento poseen un argumento del mismo tipo, a continuación realizaremos su derivación aislada correspondiente. De esta manera, en las posteriores demostraciones solo tendremos que referirnos a la siguiente prueba para evitar su repetición. Notar que en esta instancia del programa, como se mencionó a lo largo de la sección, se permite la introducción de variables de tipo e identificadores para tamaños de arreglos.

Prueba 4. Derivación de corrección para el tipo *list*.

$$\dfrac{(list, \{A\}, \textbf{pointer } node \textbf{ of } A) \in \pi_s \qquad \dfrac{T \in \langle typevariable\rangle_{tv}}{\emptyset_e, \pi_s, \pi_t \overset{\langle typevariable\rangle_{tv}, \langle sname\rangle_{sn}}{\vdash_t} T}\ (6)}{\emptyset_e, \pi_s, \pi_t \overset{\langle typevariable\rangle_{tv}, \langle sname\rangle_{sn}}{\vdash_t} list \textbf{ of } T}\ (10)$$

Como la derivación de corrección de una función puede ser extensa, asumiremos que existe una derivación `D` del juicio de tipado para el cuerpo de la función, dado que todavía no hemos introducido reglas de tipado para estas construcciones sintácticas; las cuales como ya mencionamos serán presentadas en la siguiente sección. Más adelante, una vez que se hayan detallado las reglas apropiadas, presentaremos la derivación del juicio. Utilizando la derivación anterior solo es necesario verificar el tipo del retorno de la función, lo cual es inmediato.

Prueba 5. Derivación de corrección para la función *isEmpty*.

$$\dfrac{\dfrac{\text{Prueba 4}}{\pi_{\mathbf{T}} \overset{\pi_{tv}, \pi_{sn}}{\vdash_t} list \textbf{ of } T}\ (10) \qquad \dfrac{}{\pi_{\mathbf{T}} \overset{\{T\}_{tv}, \emptyset_{sn}}{\vdash_t} \textbf{bool}}\ (1) \qquad \dfrac{\texttt{D}}{\pi_{\mathbf{T}}, \pi_f, \emptyset_p, \{T\}_{tv}, \emptyset_{sn} \vdash_b^{isEmpty} body}\ (22)}{\pi_{\mathbf{T}}, \emptyset_f, \emptyset_p \vdash_{fp} \textbf{fun } isEmpty\ (l : list \textbf{ of } T) \textbf{ ret } b : \textbf{bool in } body : \pi_f, \emptyset_p}\ (19)$$

donde $\pi_{tv} = \langle typevariable\rangle_{tv}$, y $\pi_{sn} = \langle sname\rangle_{sn}$.

El fragmento de código declarado (Código 4.2) comprende la implementación, en la sintaxis concreta del lenguaje, de la función verificada. La representación de la lista vacía la hacemos mediante el puntero `null`. Notar que en la prueba se utilizó una metavariable para representar al cuerpo de la construcción. Con esta abstracción, logramos concentrarnos en los aspectos relevantes para el análisis del prototipo de la función.

```
1 fun isEmpty (l : list of (T)) ret b : bool
2   b := l == null
3 end fun
```

Código 4.2: Función *isEmpty* para Lista

De forma análoga al caso anterior, asumiremos existe una derivación `D` del juicio de tipado para el cuerpo. Notar que hemos acumulado la información obtenida en el juicio de tipado previo, a pesar que la misma no es utilizada durante la aplicación de las reglas. Esto pone de manifiesto la importancia en el orden de declaración para las funciones y los procedimientos de un programa. En esta derivación particular, solo se debe verificar el tipo del único parámetro del procedimiento.

Prueba 6. Derivación de corrección para el procedimiento *tail*.

$$\dfrac{\dfrac{\text{Prueba 4}}{\pi_{\mathbf{T}} \overset{\pi_{tv}, \pi_{sn}}{\vdash_t} list \textbf{ of } T}\ (10) \qquad \dfrac{\texttt{D}}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \{T\}_{tv}, \emptyset_{sn} \vdash_b^{tail} body}\ (22)}{\pi_{\mathbf{T}}, \pi_f, \emptyset_p \vdash_{fp} \textbf{proc } tail\ (\textbf{in/out } l : list \textbf{ of } T) \textbf{ in } body : \pi_f, \pi_p}\ (20)$$

donde $\pi_{tv} = \langle typevariable\rangle_{tv}$, y $\pi_{sn} = \langle sname\rangle_{sn}$.

La implementación del procedimiento verificado, se ilustra en el código (Código 4.3). El comentario en el fragmento de código indica la precondición del procedimiento, el cual no puede ser invocado con una lista vacía, ya que se asume que la lista tiene al menos un elemento. Para eliminar al primer elemento de una lista es necesario liberar la memoria reservada para su respectivo nodo, donde previamente se modificó la lista para que señale al sucesor del elemento.

```
{- PRE: !isEmpty(l) -}
proc tail (in/out l : list of (T))
  var p : pointer of node of (T)
  p := l
  l := l->next
  free(p)
end proc
```

Código 4.3: Procedimiento *tail* para Lista

4.3.6. Chequeos para Cuerpos de Funciones y Procedimientos

Una parte importante en la derivación del *juicio de tipado* para la declaración de una función, o un procedimiento, es dar la derivación para su respectivo cuerpo. Lo cual implica verificar las declaraciones de variables junto con las sentencias que lo conforman. Para las primeras, se deberá garantizar la unicidad de los identificadores empleados para nombrar variables, además de comprobar la validez de los tipos que las definen. Adicionalmente las sentencias tendrán su propio conjunto de reglas para efectuar su análisis. Según la sintaxis del lenguaje, el cuerpo *body* de alguna de las construcciones anteriores posee la siguiente forma, donde $n \geq 0$ y $m > 0$; lo cual implica que el cuerpo de una función, o un procedimiento, tiene al menos una sentencia.

$$
\begin{gathered}
variabledecl_1 \\
\dots \\
variabledecl_n \\
sentence_1 \\
\dots \\
sentence_m
\end{gathered}
$$

En esta etapa del análisis se puede observar que contamos con una gran cantidad de información contextual, en nuestro alcance, para el chequeo del cuerpo de determinada función o procedimiento. Disponemos de los tipos definidos, las funciones y procedimientos declarados, e incluso las variables, de tipo y de tamaño, introducidas en el prototipo de la actual función, o procedimiento, siendo verificada. Por lo tanto para facilitar la lectura de las siguientes reglas, flexibilizaremos un poco la notación para permitir el uso de determinados contextos como si fuesen funciones.

$$
\pi_{\mathbf{FP}}(fp) = \begin{cases} (\{fa_1, \dots, fa_l\}, fr, cs) & \text{si } (fp, \{fa_1, \dots, fa_l\}, fr, cs) \in \pi_f \\ (\{pa_1, \dots, pa_l\}, cs) & \text{si } (fp, \{pa_1, \dots, pa_l\}, cs) \in \pi_p \end{cases}
$$

Haciendo un abuso de notación diremos que vale $x \in \pi_{\mathbf{FP}}(fp)$ cuando este identificador haya sido introducido como parámetro en el prototipo de fp.

Contexto para Variables

La declaración de variables dentro del cuerpo de funciones o procedimientos, nos obliga a utilizar un contexto adicional para almacenar los identificadores introducidos, junto con sus correspondientes tipos. Similar como sucede con otros contextos, es necesario garantizar como invariante la unicidad de los nombres de variables que se almacenan en este conjunto. Abusando la notación, diremos que $x \in \pi_v$ cuando una variable ya se encuentra declarada en el contexto.

$$
\pi_v \subseteq \{(x, \theta) \mid x \in \langle id \rangle \wedge \theta \in \langle type \rangle\}
$$

Para la declaración de una nueva variable x en una función o un procedimiento fp, es fundamental asegurar que no exista una declaración con igual identificador, es decir $x \notin \pi_v$. Además hay que tener la precaución que su nombre tampoco aparezca en el prototipo de la función o el procedimiento donde ocurre la declaración, es decir $x \notin \pi_{\mathbf{FP}}(fp)$. Por lo tanto, escribiremos $x \notin \pi_{\mathbf{FP}}(fp) \cup \pi_v$ para decir que el identificador x es fresco.

Regla para Declaración de Variables

El siguiente *juicio de tipado* denota que es válido extender al contexto de variables π_v con una nueva variable, bajo todos los contextos ya mencionados. Notar el uso del índice fp para representar al identificador de la función, o el procedimiento, que encapsula a la declaración.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{tv}, \pi_{sn}, \pi_v \vdash^{fp}_{vd} variabledecl : \pi'_v$$

Para determinar que una declaración de variables dentro del cuerpo de una función o un procedimiento es válida, hay que verificar dos propiedades. En primer lugar, todas las variables que se intentan declarar deben ser frescas. Como se mencionó recientemente, hay que garantizar que ningún identificador haya sido introducido aún en el alcance actual. En segundo lugar, el tipo asociado a las variables debe ser válido. Solo se permite el uso de los identificadores para tamaños de arreglos, y las variables de tipo, que hayan sido especificados previamente en el prototipo. Recordar que la invariante de construcción del contexto de variables imponía la unicidad de los identificadores que lo conforman. Por lo que de forma implícita, nos aseguramos que todas las variables definidas son distintas entre sí.

Regla para Función/Procedimiento 21. Declaración de Variables

$$\frac{x_i \notin \pi_{\mathbf{FP}}(fp) \cup \pi_v \qquad \pi_{\mathbf{T}} \overset{\pi_{tv},\pi_{sn}}{\vdash_t} \theta}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{tv}, \pi_{sn}, \pi_v \vdash^{fp}_{vd} \mathbf{var}\ x_1, \ldots, x_l : \theta : \pi'_v}$$

donde $\pi'_v = (x_1, \theta) \triangleright \ldots \triangleright (x_l, \theta) \triangleright \pi_v$.

Ejemplo de Prueba

Continuando con la derivación del fragmento (Código 4.3), demostraremos la corrección de la declaración de variable presente en el procedimiento. Recordar que para chequear el cuerpo del procedimiento *tail* contamos con los contextos previamente definidos, y en particular, con la declaración de los tipos *list* y *node*. Notar que el único identificador introducido en el alcance actual, es el del parámetro de entrada/salida l, por lo que vale $\pi_{\mathbf{FP}}(tail) \cup \emptyset_v = \{l\}$.

Prueba 7. Derivación de corrección para declaración de variable.

$$\dfrac{p \notin \{l\} \qquad \dfrac{\dfrac{(node, \{Z\}, \{elem : Z, next : \mathbf{pointer}\ node\ \mathbf{of}\ Z\}) \in \pi_t \qquad \dfrac{T \in \{T\}}{\emptyset_e, \pi_s, \pi_t \overset{\{T\},\emptyset_{sn}}{\vdash_t} T}\ (6)}{\emptyset_e, \pi_s, \pi_t \overset{\{T\},\emptyset_{sn}}{\vdash_t} node\ \mathbf{of}\ T}\ (11)}{\emptyset_e, \pi_s, \pi_t \overset{\{T\},\emptyset_{sn}}{\vdash_t} \mathbf{pointer}\ node\ \mathbf{of}\ T}\ (2)}{\emptyset_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \{T\}_{tv}, \emptyset_{sn}, \emptyset_v \vdash^{tail}_{vd} \mathbf{var}\ p : \mathbf{pointer}\ node\ \mathbf{of}\ T : \pi_v}\ (21)$$

donde $\pi_v = \{(p, \mathbf{pointer}\ node\ \mathbf{of}\ T)\}$.

Regla para Cuerpos de Funciones y Procedimientos

El *juicio de tipado* correspondiente al cuerpo de una función o un procedimiento fp, define su corrección bajo las declaraciones de tipo contenidas en $\pi_{\mathbf{T}}$, las declaraciones de funciones y procedimientos contenidas en $\pi_{\mathbf{FP}}$, y los elementos introducidos en el prototipo contenidos en π_{tv} y π_{sn}. En esta verificación no será necesaria la extensión de ningún contexto, ya que toda la información dentro del cuerpo es local al prototipo que lo encapsula.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{tv}, \pi_{sn} \vdash_b^{fp} body$$

Dar una derivación para este juicio de tipado consiste en dar derivaciones para cada una de las declaraciones de variables, donde el contexto inicial para el chequeo de una declaración será el contexto extendido producto de una declaración previa. Además utilizando el contexto extendido con todas las declaraciones de variables, tenemos que dar derivaciones para sus sentencias. El juicio de tipado para las sentencias y sus respectivas reglas será formalizado más adelante, una vez que se haya precisado la manera de resolver el polimorfismo que admiten las funciones y los procedimientos del programa.

Regla para Función/Procedimiento 22. Cuerpo

$$\frac{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{tv}, \pi_{sn}, \pi_v^{i-1} \vdash_{vd}^{fp} vd_i : \pi_v^i \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v^n \vdash_s^{fp} s_j}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{tv}, \pi_{sn} \vdash_b^{fp} vd_1 \dots vd_n \quad s_1 \dots s_m}$$

donde el contexto inicial de variables es vacío $\pi_v^0 = \emptyset$.

4.3.7. Operación de Sustitución

Los tipos que contienen ocurrencias de variables de tipo, o identificadores para tamaños de arreglos, pueden ser instanciados mediante alguna sustitución. Intuitivamente la aplicación de una sustitución a un tipo θ se propaga por toda la estructura del mismo, salvo cuando se encuentra con una variable, de tipo o de tamaño, en cuyo caso la substituye según lo indicado por la sustitución $\delta \in \Delta$.

$$\_ \mid \_ \in \langle type \rangle \times \Delta \to \langle type \rangle$$
$$\theta \mid \delta \in \langle type \rangle$$

Sustitución de Variables de Tipo

Un tipo puede ser instanciado sustituyendo sus variables por otros tipos particulares. Lo cual puede ocurrir cuando tenemos o bien, una declaración de tipo con parámetros que permite definir tipos más precisos instanciando algunas (o todas) sus variables de tipo; o una definición de función o procedimiento polimórfica. Esto implica que al aplicar la función o el procedimiento, será necesario igualar los tipos de los parámetros esperados contra los tipos de los argumentos recibidos mediante una sustitución. Definimos al conjunto Δ_{tv} de todas las sustituciones de variables de tipo en tipos, junto con la notación utilizada para representar una sustitución finita.

$$\Delta_{tv} = \langle typevariable \rangle \to \langle type \rangle$$
$$[tv_1 : \theta_1, \dots, tv_l : \theta_l]_{tv} \in \Delta_{tv}$$

El operador de aplicación de sustitución aplica la sustitución provista δ_{tv} a todas las variables de tipo. Como es esperable no realiza ninguna modificación a los tipos básicos, y para el caso de los tipos más complejos, simplemente tenemos que propagar la aplicación del operador.

$$\begin{aligned}
\delta_{tv} &\in \Delta_{tv} \\
\textbf{int} \mid \delta_{tv} &= \textbf{int} \\
\textbf{real} \mid \delta_{tv} &= \textbf{real} \\
\textbf{bool} \mid \delta_{tv} &= \textbf{bool} \\
\textbf{char} \mid \delta_{tv} &= \textbf{char} \\
\textbf{pointer}\ \theta \mid \delta_{tv} &= \textbf{pointer}\ (\theta \mid \delta_{tv}) \\
\textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ \theta \mid \delta_{tv} &= \textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ (\theta \mid \delta_{tv}) \\
tv \mid \delta_{tv} &= \delta_{tv}(tv) \\
tn \mid \delta_{tv} &= tn \\
tn\ \textbf{of}\ \theta_1, \ldots, \theta_n \mid \delta_{tv} &= tn\ \textbf{of}\ (\theta_1 \mid \delta_{tv}), \ldots, (\theta_n \mid \delta_{tv})
\end{aligned}$$

Sustitución de Identificadores de Tamaños

De forma análoga a la sustitución anterior, un tipo que posee ocurrencias de identificadores para tamaños de arreglos podrá ser instanciado reemplazando estos elementos por otros tamaños particulares. Se define al conjunto de todas las sustituciones de identificadores para tamaños en tamaños, de la siguiente forma.

$$\Delta_{sn} = \langle sname \rangle \to \langle arraysize \rangle$$

La operación de sustitución para identificadores de tamaños opera de manera análoga a la de sustitución para variables de tipos. Aunque es importante remarcar que la operación se debe propagar a los tamaños de las dimensiones de arreglos que ocurren en el tipo. De esta manera si el tamaño as es variable, deberá ser reemplazado según lo determinado por δ_{sn}. En caso contrario no se realizará ninguna modificación al mismo.

$$\begin{aligned}
\delta_{sn} &\in \Delta_{sn} \\
\textbf{int} \mid \delta_{sn} &= \textbf{int} \\
\textbf{real} \mid \delta_{sn} &= \textbf{real} \\
\textbf{bool} \mid \delta_{sn} &= \textbf{bool} \\
\textbf{char} \mid \delta_{sn} &= \textbf{char} \\
\textbf{pointer}\ \theta \mid \delta_{sn} &= \textbf{pointer}\ (\theta \mid \delta_{sn}) \\
\textbf{array}\ as_1, \ldots, as_n\ \textbf{of}\ \theta \mid \delta_{sn} &= \textbf{array}\ (as_1 \mid \delta_{sn}), \ldots, (as_n \mid \delta_{sn})\ \textbf{of}\ (\theta \mid \delta_{sn}) \\
tv \mid \delta_{sn} &= tv \\
tn \mid \delta_{sn} &= tn \\
tn\ \textbf{of}\ \theta_1, \ldots, \theta_n \mid \delta_{sn} &= tn\ \textbf{of}\ (\theta_1 \mid \delta_{sn}), \ldots, (\theta_n \mid \delta_{sn})
\end{aligned}$$

$$as \mid \delta_{sn} = \begin{cases} as & \text{si } as \in \langle natural \rangle \\ \delta_{sn}(as) & \text{si } as \in \langle sname \rangle \end{cases}$$

4.3.8. Instancias de Clases

En *Haskell*, una clase puede ser pensada como una especie de interfaz que define algún comportamiento. Se dice que un tipo tiene una *instancia* de la clase cuando soporta e implementa el comportamiento que esta clase describe. Recordamos que en nuestro lenguaje tenemos un sistema de clases e instancias con el mismo espíritu, pero mucho más restringido ya que no permitimos declarar nuevas clases ni instancias.

A continuación describiremos entonces, de manera informal, cuando un tipo tiene instancia de alguna clase. Actualmente en el lenguaje se encuentran definidas de forma nativa solo dos clases, **Eq** y **Ord**. En base a la categoría de un tipo, y al entorno en el que se encuentra situado, habrá distintas condiciones para que el mismo pueda satisfacer, o no, una determinada clase del lenguaje. Diremos informalmente que un tipo θ satisface una clase cl cuando:

1. Si es un tipo básico, es decir $\theta \in \{\mathbf{int}, \mathbf{real}, \mathbf{char}, \mathbf{bool}\}$, entonces satisface naturalmente ambas clases **Eq**, y **Ord**. Las operaciones de orden y de igualdad que ofrecen estas clases, se definen de la manera habitual.

2. En el cuerpo de una función o un procedimiento fp, una variable de tipo tv satisface una determinada clase cl_i, solo si en el prototipo correspondiente se impone como restricción. Es decir, en cs ocurre la restricción de clases $tv : cl_1, \ldots, cl_i, \ldots, cl_m$. Las operaciones son determinadas de acuerdo al tipo concreto que la variable asume durante la ejecución.

3. Si es un tipo enumerado tn, definido de la forma **enum** $tn = cn_1, \ldots, cn_m$, entonces satisface ambas clases **Eq**, y **Ord**. La instancia de igualdad se define como $cn_i = cn_i$ para todo i, y la de orden como $cn_i < cn_j$, donde vale que $1 \leq i < j \leq m$.

4. Si es un sinónimo de tipo tn, definido de la forma **syn** $tn = \theta$, entonces va a satisfacer las mismas clases que el tipo de su definición. Las operaciones de orden e igualdad serían las que implementa θ.

5. Si es un sinónimo de tipo con parámetros tn **of** $\theta_1, \ldots, \theta_l$, definido de la forma **syn** tn **of** $tv_1, \ldots, tv_l = \theta$, entonces va a satisfacer las mismas clases que el tipo al que representa. El cual se obtiene de aplicar la sustitución finita adecuada $\theta \mid [tv_1 : \theta_1, \ldots, tv_l : \theta_l]_{tv}$. Las operaciones disponibles del sinónimo, son las definidas por este último tipo.

4.3.9. Chequeos para Sentencias

En esta sección cubriremos las reglas de chequeo para las sentencias del lenguaje donde queremos asegurar propiedades como la correcta asignación de expresiones a variables, la especificación de límites válidos para la sentencia *for*, el empleo de guardas de tipo booleano para las sentencias *if* y *while*, entre otras. En particular, para la llamada a procedimientos haremos uso de los últimos conceptos introducidos sobre la operación de sustitución, y las instancias de clases.

Reglas para Sentencias

Para el *juicio de tipado* de una sentencia, tenemos a nuestro alcance los contextos sobre los tipos definidos ($\pi_{\mathbf{T}}$), las funciones y los procedimientos declarados ($\pi_{\mathbf{FP}}$), los identificadores para tamaños de arreglos introducidos en el prototipo (π_{sn}), y las variables introducidas previamente en el cuerpo (π_v). Prescindimos de la información sobre las variables de tipo detalladas en el encabezado, ya que no será necesaria para el siguiente análisis. Para abreviar la notación empleada, utilizaremos $\pi_{\mathbf{V}}$ para representar al conjunto de variables, y al de identificadores para

tamaños de arreglos. Adicionalmente haremos referencia al contexto $\pi_{\mathbf{P}}$ para simbolizar a toda la información necesaria para efectuar el análisis.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_s^{fp} sentence$$
$$\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_s^{fp} sentence$$
$$\pi_{\mathbf{P}} \vdash_s^{fp} sentence$$

El análisis de la sentencia *skip* es trivial. Su verificación es inmediata, ya que la regla no presenta ninguna premisa.

Regla para Sentencias 23. Skip

$$\frac{}{\pi_{\mathbf{P}} \vdash_s^{fp} \mathbf{skip}}$$

La regla de la asignación presenta la primer situación interesante en el análisis de las sentencias. Se deben verificar sus dos componentes, la variable a modificar y la expresión que se le asignará, para asegurar que ambas poseen el mismo tipo. Notar que se introduce un nuevo juicio de tipado que será detallado más adelante, el cual nos asegura que una expresión tiene algún tipo bajo determinados contextos.

Regla para Sentencias 24. Asignación

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} v : \theta \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e : \theta}{\pi_{\mathbf{P}} \vdash_s^{fp} v := e}$$

La tarea del procedimiento **alloc** es reservar memoria, y por lo tanto es necesario que su argumento tenga tipo puntero. Recordar que la sintaxis del lenguaje solo permite la escritura de variables como argumento en la llamada del procedimiento.

Regla para Sentencias 25. Alloc

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} v : \mathbf{pointer}\ \theta}{\pi_{\mathbf{P}} \vdash_s^{fp} \mathbf{alloc}\ v}$$

Para el procedimiento **free**, ocurre una situación similar. Es necesario verificar que el argumento recibido sea efectivamente un puntero. En este caso, la sentencia se encarga de liberar el espacio de memoria que referencia el puntero.

Regla para Sentencias 26. Free

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} v : \mathbf{pointer}\ \theta}{\pi_{\mathbf{P}} \vdash_s^{fp} \mathbf{free}\ v}$$

En el caso de las sentencias *if* y *while* tenemos que asegurar que el tipo de la expresión en la guarda sea booleano, y además de tener derivaciones para las sentencias de sus cuerpos.

Regla para Sentencias 27. While

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_s^{fp} s_i}{\pi_{\mathbf{P}} \vdash_s^{fp} \mathbf{while}\ e\ \mathbf{do}\ s_1 \ldots s_m}$$

Regla para Sentencias 28. If

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_s^{fp} s_i \qquad \pi_{\mathbf{P}} \vdash_s^{fp} s'_j}{\pi_{\mathbf{P}} \vdash_s^{fp} \mathbf{if}\ e\ \mathbf{then}\ s_1 \ldots s_m\ \mathbf{else}\ s'_1 \ldots s'_n}$$

La sentencia *for* presenta dos construcciones sintácticas distintas. Debido que la sentencia declara de forma implícita una variable, es necesario verificar que el identificador introducido sea fresco con respecto al alcance actual. Adicionalmente hay que chequear que el tipo de los límites sea entero. Finalmente se debe analizar la secuencia de sentencias que forman al cuerpo de la iteración. Notar que la variable declarada es local a este último bloque de sentencias.

Regla para Sentencias 29. For To

$$\frac{x \notin \pi_{\mathbf{FP}}(fp) \cup \pi_v \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_e^{fp} e_i : \mathbf{int} \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi'_v \vdash_s^{fp} s_i}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_s^{fp} \mathbf{for}\ x := e_1\ \mathbf{to}\ e_2\ \mathbf{do}\ s_1 \ldots s_m}$$

donde $\pi'_v = (x, \mathbf{int}) \triangleright \pi_v$.

Regla para Sentencias 30. For Downto

$$\frac{x \notin \pi_{\mathbf{FP}}(fp) \cup \pi_v \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_e^{fp} e_i : \mathbf{int} \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi'_v \vdash_s^{fp} s_i}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_s^{fp} \mathbf{for}\ x := e_1\ \mathbf{downto}\ e_2\ \mathbf{do}\ s_1 \ldots s_m}$$

donde $\pi'_v = (x, \mathbf{int}) \triangleright \pi_v$.

La regla para la llamada de procedimientos es la más compleja de todas las presentadas hasta el momento para las sentencias. Será necesario verificar una serie de propiedades relacionadas con el polimorfismo que admite el procedimiento; para las cuales utilizaremos los conceptos previos sobre la operación de sustitución, y la definición informal sobre cuando un tipo tiene instancia de cierta clase. La operación pretende hacer coincidir los tipos esperados por los parámetros del procedimiento, contra los tipos de los argumentos recibidos en la respectiva llamada. Por lo cual es necesario encontrar una función de sustitución para las variables de tipo, y otra para los identificadores de tamaños, que se especifican en el prototipo de la declaración. Claramente no siempre será posible establecer una igualdad entre los tipos esperados y los tipos recibidos, por lo que la existencia de las funciones de sustitución, será lo que buscamos para verificar la validez de la llamada al procedimiento. Denominaremos a esta operación *unificación*.

Con la función de sustitución para identificadores de tamaños, se intenta reemplazar todas las variables de tamaño que ocurren en los parámetros de la declaración, para hacerlas coincidir con los tamaños actuales de los argumentos de la llamada. De forma análoga, con la función de sustitución para variables de tipo se pretende igualar todos las variables de tipo especificadas en el encabezado del procedimiento, con los tipos actuales de las expresiones recibidas como argumento de la llamada. Notar que la idea es encontrar la existencia de las funciones de sustitución, lo cual nos asegura que existe la unificación que buscamos.

El polimorfismo paramétrico que admite un procedimiento, puede ser refinado mediante la especificación de restricciones de clases para las variables de tipo introducidas en el prototipo. En base a la función de sustitución para variables de tipo obtenida en la unificación, debemos verificar que todas las restricciones sean satisfechas. El tipo que adopta una determinada variable, luego de aplicada la sustitución, tendrá que ser instancia de todas las clases que se le imponen de acuerdo a las reglas especificadas previamente.

La regla entonces tendrá cuatro premisas. La primera, debe asegurar que el procedimiento esta definido y adicionalmente que la cantidad de argumentos en la llamada coincida con el número de parámetros en la declaración. La segunda y tercera, prueban que es posible unificar los tipos esperados por el procedimiento con los tipos actuales de las expresiones, mediante la aplicación de alguna función de sustitución. La cuarta, comprueba que se satisfacen las restricciones de clases impuestas para las variables de tipo de la declaración. Notar que si el procedimiento definido no introduce ningún elemento polimórfico en su prototipo, es decir variables de tipo o identificadores para tamaños de arreglos, entonces no será necesario probar la existencia de ninguna función de sustitución; por lo que en esta situación solo será necesario comprobar la igualdad entre los tipos de los argumentos y los parámetros.

Regla para Sentencias 31. Procedimientos

$$\frac{(p, \{io_1\ a_1 : \theta_1^*, \ldots, io_n\ a_n : \theta_n^*\}, cs) \in \pi_p \qquad \pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e^{fp} e_i : \theta_i \qquad \text{(Unif.)} \qquad \text{(Rest.)}}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_s^{fp} p(e_1, \ldots, e_n)}$$

Donde existe una sustitución tal que, los tipos de los parámetros esperados se unifican con los tipos de los argumentos recibidos.

$$\exists \delta_{sn} \in \Delta_{sn}, \delta_{tv} \in \Delta_{tv}.\ \forall i \in \{1 \ldots n\}.\ \theta_i^* \mid \delta_{sn} \mid \delta_{tv} = \theta_i \qquad \text{(Unif.)}$$

Si esta sustitución existe, entonces se deben satisfacer todas las restricciones de clase impuestas en el prototipo del procedimiento.

$$\forall (tv : cl_1, \ldots, cl_m) \in cs.\ \delta_{tv}(tv) \text{ satisface las clases } cl_1, \ldots, cl_m \qquad \text{(Rest.)}$$

Ejemplo de Sustitución

Debido a la complejidad de la regla, ilustraremos la aplicación de una sustitución con un breve ejemplo. Supongamos que hemos definido un algoritmo de ordenación. El fragmento (Código 4.4) se encargará de permutar los valores en dos posiciones válidas de un arreglo. Mientras, en el código (Código 4.5) se aplica la ordenación *por selección* a un arreglo. Notar que ambos procedimientos tienen como parámetro un arreglo con tamaño a definir de forma dinámica, y que los valores almacenados podrán tener cualquier tipo; en el caso de la implementación para la ordenación *por selección*, deberá ser instancia de la clase **Ord**. La abstracción permite definir un algoritmo de ordenación para arreglos, que será independiente del tamaño y del tipo de valores almacenados por esta estructura.

```
1 {- PRE: 1 <= i, j <= m -}
2 proc swap (in/out a : array [m] of T, in i, j : int)
3   var temp : T
4   temp := a[i]
5   a[i] := a[j]
6   a[j] := temp
7 end proc
```

Código 4.4: Permutación de Valores

```
proc selectionSort (in/out a : array [n] of T)
where (T : Ord)
  var minP : int
  for i := 1 to n do
    minP := i
    for j := i + 1 to n do
      if a[j] < a[minP] then minP := j fi
    od
    swap(a, i, minP)
  od
end proc
```

Código 4.5: Ordenación por Selección

Durante el proceso de ordenación, se aplican las permutaciones de valor adecuadas a medida que se avanza en la ejecución del código. En el procedimiento principal, se realiza la llamada `swap(a,i, minP)` una vez que se encuentra al valor, ubicado en la posición `minP`, que correspondería a la posición `i` en la ordenación definitiva del arreglo `a`. Para probar la corrección de esta sentencia, es necesario demostrar la existencia de las sustituciones δ_{sn} y δ_{tv}, las cuales permiten unificar los tipos esperados por el procedimiento contra los tipos recibidos en la llamada. Notar que ambos índices son de tipo entero, por lo que solo debemos concentrarnos en igualar el tipo de los arreglos. Sea $\delta_{sn}(m) = n$ y $\delta_{tv}(T) = T$, entonces mediante su aplicación obtenemos la unificación deseada.

$$\textbf{array } m \textbf{ of } T \mid \delta_{sn} \mid \delta_{tv} = \textbf{array } n \textbf{ of } T$$
$$\textbf{int} \mid \delta_{sn} \mid \delta_{tv} = \textbf{int}$$

Supongamos que en el alcance actual se encuentra declarado un arreglo de enteros, con tamaño diez, representado con el identificador `a'`. Para aplicar el algoritmo de ordenación sobre el arreglo, es necesario realizar la llamada `selectionSort(a')`. Puesto que el procedimiento especifica una restricción de clase para el tipo de los valores almacenados en el arreglo, hay una condición adicional que satisfacer. En este caso para probar la validez de la sentencia, debemos demostrar la existencia de las sustituciones δ_{sn} y δ_{tv}, necesarias para efectuar la unificación adecuada, sumado a verificar que los elementos del arreglo pueden ser ordenados. Sea $\delta_{sn}(n) = 10$ y $\delta_{tv}(T) = \textbf{int}$, entonces mediante su aplicación obtenemos la unificación deseada y se satisface trivialmente la restricción de clase.

$$\textbf{array } n \textbf{ of } T \mid \delta_{sn} \mid \delta_{tv} = \textbf{array } 10 \textbf{ of int}$$
$$\delta_{tv}(T) \text{ satisface la clase } \textbf{Ord}$$

4.3.10. Chequeos para Expresiones

La notación utilizada para el análisis de expresiones ya fue introducida en la sección previa, de todas formas, a continuación describiremos su significado formalmente. El siguiente *juicio de tipado* denota que la expresión e tiene tipo θ bajo los contextos comprendidos por $\pi_{\mathbf{P}}$, en el cuerpo de la función o el procedimiento fp.

$$\pi_{\mathbf{P}} \vdash^{fp}_{e} e : \theta$$

Las reglas para las constantes son directas, ya que no presentan ninguna premisa. Además cualquier constante enumerada presente en el contexto correspondiente, demuestra que posee el tipo con el que fue declarada.

Regla para Expresiones 32. Valores Constantes

$$\frac{}{\pi_{\mathbf{P}} \vdash_e^{fp} n : \mathbf{int}} \qquad \frac{}{\pi_{\mathbf{P}} \vdash_e^{fp} r : \mathbf{real}}$$

$$\frac{}{\pi_{\mathbf{P}} \vdash_e^{fp} b : \mathbf{bool}} \qquad \frac{}{\pi_{\mathbf{P}} \vdash_e^{fp} c : \mathbf{char}}$$

Regla para Expresiones 33. Infinito

$$\frac{}{\pi_{\mathbf{P}} \vdash_e^{fp} \mathbf{inf} : \mathbf{int}}$$

Regla para Expresiones 34. Constantes Enumeradas

$$\frac{(tn, \{cn_1, \ldots, cn_i, \ldots, cn_m\}) \in \pi_e}{\pi_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{fp} cn_i : tn}$$

La constante **null** simboliza un puntero que direcciona a una posición inválida de memoria. Por lo tanto, la constante puede ser utilizada como un puntero que señala a un tipo particular cualquiera.

Regla para Expresiones 35. Puntero Nulo

$$\frac{}{\pi_{\mathbf{P}} \vdash_e^{fp} \mathbf{null} : \mathbf{pointer}\ \theta}$$

Una variable puede haber sido introducida de dos maneras: mediante la declaración en el cuerpo de una función o un procedimiento, o como parámetro de alguna de estas construcciones. Definimos tres reglas que nos permitirán deducir el tipo de una variable en base al contexto correspondiente. En el caso de los identificadores para tamaños de arreglos ocurre algo similar, pero solo pueden ser introducidos en el prototipo de funciones o procedimientos, y su tipo debe ser siempre entero.

Regla para Expresiones 36. Variables Declaradas

$$\frac{(x, \theta) \in \pi_v}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_e^{fp} x : \theta}$$

Regla para Expresiones 37. Parámetros de Función

$$\frac{\pi_{\mathbf{FP}}(f) = (\{a_1 : \theta_1, \ldots, a_l : \theta_l\}, a_r : \theta_r, cs)}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{f} a_i : \theta_i}$$

Regla para Expresiones 38. Parámetros de Procedimiento

$$\frac{\pi_{\mathbf{FP}}(p) = (\{oi_1\ a_1 : \theta_1, \ldots, oi_l\ a_l : \theta_l\}, cs)}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{p} a_i : \theta_i}$$

Regla para Expresiones 39. Identificadores de Tamaños

$$\frac{as \in \pi_{sn}}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{sn}, \pi_v \vdash_e^{fp} as : \mathbf{int}}$$

El siguiente grupo de reglas nos permiten chequear la correcta aplicación de los operadores sobre variables. El operador $\star$ aplicado a una variable v nos permite acceder al valor del bloque de memoria apuntado por esta, por lo tanto el tipo de v debe ser un puntero.

Regla para Expresiones 40. Acceso a Punteros

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} v : \mathbf{pointer}\ \theta}{\pi_{\mathbf{P}} \vdash_e^{fp} \star v : \theta}$$

La regla para el acceso al campo de una tuplas es interesante. Recordemos que el tipo tupla no existe propiamente, sino que es una construcción sintáctica que nos permite declarar nuevos tipos. Por lo tanto, la variable a la cual le aplicamos el operador para acceder a uno de sus campos, debe haber sido declarada como una estructura de tipo tupla. De acuerdo a los parámetros declarados en la definición del tipo, y los argumentos de tipo que fueron especificados cuando se introdujo la variable, debe existir una sustitución adecuada.

Regla para Expresiones 41. Acceso a Tuplas

$$\frac{\pi_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{fp} v : tn\ \mathbf{of}\ \theta_1, \ldots, \theta_l \qquad (tn, \{tv_1, \ldots, tv_l\}, \{fn_1 : \theta_1^*, \ldots, fn_m : \theta_m^*\}) \in \pi_t}{\pi_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{fp} v.fn_i : \theta_i^* \mid [tv_1 : \theta_1, \ldots, tv_l : \theta_l]_{tv}}$$

La regla para el acceso a valores de un arreglo debe comprobar que la variable involucrada sea en efecto un arreglo. Además todas las expresiones que determinan cual es la posición que se quiere acceder deben tener tipo entero, y su cantidad tiene que coincidir con las dimensiones que posee la estructura. Notar que no es posible realizar un acceso parcial de un arreglo.

Regla para Expresiones 42. Acceso a Arreglos

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} v : \mathbf{array}\ as_1, \ldots, as_n\ \mathbf{of}\ \theta \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e_i : \mathbf{int}}{\pi_{\mathbf{P}} \vdash_e^{fp} v[e_1, \ldots, e_n] : \theta}$$

El siguiente conjunto de reglas define los chequeos intuitivos ha realizar para los operadores aritméticos y booleanos. Notar que los operadores numéricos pueden ser utilizados para trabajar con valores enteros y reales.

Regla para Expresiones 43. Operadores Binarios Numéricos

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 : \mathbf{int} \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e_2 : \mathbf{int}}{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 \oplus e_2 : \mathbf{int}} \qquad \frac{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 : \mathbf{real} \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e_2 : \mathbf{real}}{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 \oplus e_2 : \mathbf{real}}$$

donde $\oplus \in \{+, -, *, /, \%\}$.

Regla para Expresiones 44. Operadores Unarios Numéricos

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e : \mathbf{int}}{\pi_{\mathbf{P}} \vdash_e^{fp} -e : \mathbf{int}} \qquad \frac{\pi_{\mathbf{P}} \vdash_e^{fp} e : \mathbf{real}}{\pi_{\mathbf{P}} \vdash_e^{fp} -e : \mathbf{real}}$$

Regla para Expresiones 45. Operadores Binarios Booleanos

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e_2 : \mathbf{bool}}{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 \oplus e_2 : \mathbf{bool}}$$

donde $\oplus \in \{\&\&, ||\}$.

Regla para Expresiones 46. Operadores Unarios Booleanos

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e : \mathbf{bool}}{\pi_{\mathbf{P}} \vdash_e^{fp} !e : \mathbf{bool}}$$

Para el caso de las operaciones de igualdad y orden, sus operandos deben ser del mismo tipo. Más importante aún existe una condición informal que requerimos, dado que aún no contamos con una definición formal del concepto de instancia, donde θ debe tener instancia **Eq** u **Ord** según corresponda.

Regla para Expresiones 47. Operadores de Igualdad

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 : \theta \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e_2 : \theta}{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 \oplus e_2 : \mathbf{bool}}$$

donde $\oplus \in \{==, != \}$, y se satisface que el tipo θ tiene instancia **Eq**.

Regla para Expresiones 48. Operadores de Orden

$$\frac{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 : \theta \qquad \pi_{\mathbf{P}} \vdash_e^{fp} e_2 : \theta}{\pi_{\mathbf{P}} \vdash_e^{fp} e_1 \oplus e_2 : \mathbf{bool}}$$

donde $\oplus \in \{<, >, <=, >=\}$, y se satisface que el tipo θ tiene instancia **Ord**.

La regla para la aplicación de una función puede considerarse como la más compleja de las reglas para expresiones, de manera similar a lo que ocurría en el caso de las sentencias con la llamada a un procedimiento; las cuales poseen una fuerte similitud entre sí. La regla posee cuatro premisas. La primera, debe asegurar la existencia de la función y que la cantidad de argumentos sea adecuada. La segunda y tercera, prueban la existencia de una sustitución que permita unificar los tipos en cuestión. La cuarta, comprueba que se satisfacen las restricciones de clases que fueron declaradas en el prototipo de la función.

Regla para Expresiones 49. Funciones

$$\frac{(f, \{a_1 : \theta_1^*, \ldots, a_n : \theta_n^*\}, a_r : \theta_r^*, cs) \in \pi_f \qquad \pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e^{fp} e_i : \theta_i \qquad \text{(Unif.)} \qquad \text{(Rest.)}}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e^{fp} f(e_1, \ldots, e_n) : \theta_r}$$

Donde existe una sustitución tal que, los tipos de los parámetros esperados se unifican con los tipos de los argumentos recibidos.

$$\exists \delta_{sn} \in \Delta_{sn}, \delta_{tv} \in \Delta_{tv}.\ (\forall i \in \{1 \ldots n\}.\ \theta_i^* \mid \delta_{sn} \mid \delta_{tv} = \theta_i) \wedge (\theta_r^* \mid \delta_{sn} \mid \delta_{tv} = \theta_r) \qquad \text{(Unif.)}$$

Si esta sustitución existe, entonces se deben satisfacer todas las restricciones de clase impuestas en el prototipo de la función.

$$\forall (tv : cl_1, \ldots, cl_m) \in cs.\ \delta_{tv}(tv) \text{ satisface las clases } cl_1, \ldots, cl_m \qquad \text{(Rest.)}$$

Ejemplo de Prueba

Retomando la demostración del fragmento (Código 4.3), podemos construir una nueva derivación para el juicio de tipado que involucra la aplicación del procedimiento *free* con la variable p. Recordar que al encontrarnos en el cuerpo del procedimiento *tail*, contamos con toda la información reunida hasta ese punto. En particular disponemos de la declaración de la *lista abstracta*, la cual comprende los tipos definidos junto con el prototipo de los operadores verificados; y los elementos introducidos en el alcance actual, como los parámetros del procedimiento y las variables declaradas en su cuerpo.

Prueba 8. Derivación de corrección para la llamada de *free*.

$$\dfrac{\dfrac{(p, \textbf{pointer } node \textbf{ of } T) \in \pi_v}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_e^{tail} p : \textbf{pointer } node \textbf{ of } T}\ (36)}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_s^{tail} \textbf{free } p}\ (26)$$

Conversión Numérica

Al conjunto de reglas actuales para expresiones, le añadiremos una regla bien general que estará definida para cualquier expresión. La cual extenderá al sistema de tipos con una versión extremadamente primitiva de subtipado; donde solo definimos la posibilidad de tipar correctamente una expresión con tipo **real**, cuando la misma sea de tipo **int**.

Regla para Expresiones 50. Conversión de **int** a **real**

$$\dfrac{\pi_{\mathbf{P}} \vdash_e^{fp} e : \textbf{int}}{\pi_{\mathbf{P}} \vdash_e^{fp} e : \textbf{real}}$$

Esta regla si bien simple nos permite, por ejemplo, realizar asignaciones de expresiones enteras a variables declaradas con tipo real. Incluso menos natural, resulta posible dar una derivación para el juicio de tipado de una asignación cuya componente izquierda tiene tipo **int** y derecha tiene tipo **real**.

$$\dfrac{\dfrac{(x, \textbf{real}) \in \pi_v}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_e^{f} x : \textbf{real}}\ (36) \qquad \dfrac{\dfrac{(y, \textbf{int}) \in \pi_v}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_e^{f} y : \textbf{int}}\ (36)}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_e^{f} y : \textbf{real}}\ (50)}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_s^{f} x := y}\ (24)$$

Así mismo, introducir esta regla es suficiente para que ahora exista la posibilidad de tener más de una derivación para un mismo juicio de tipado. Notar que para el *juicio* de la expresión $1 + 2$ existen al menos dos derivaciones distintas cuando se espera un valor real. Supongamos que los contextos comprendidos en $\pi_{\mathbf{P}}$ están definidos de forma adecuada.

$$\dfrac{\dfrac{\dfrac{}{\pi_{\mathbf{P}} \vdash_e^{fp} 1 : \textbf{int}}\ (32) \qquad \dfrac{}{\pi_{\mathbf{P}} \vdash_e^{fp} 2 : \textbf{int}}\ (32)}{\pi_{\mathbf{P}} \vdash_e^{fp} 1 + 2 : \textbf{int}}\ (43)}{\pi_{\mathbf{P}} \vdash_e^{fp} 1 + 2 : \textbf{real}}\ (50) \qquad \dfrac{\dfrac{\dfrac{}{\pi_{\mathbf{P}} \vdash_e^{fp} 1 : \textbf{int}}\ (32)}{\pi_{\mathbf{P}} \vdash_e^{fp} 1 : \textbf{real}}\ (50) \qquad \dfrac{\dfrac{}{\pi_{\mathbf{P}} \vdash_e^{fp} 2 : \textbf{int}}\ (32)}{\pi_{\mathbf{P}} \vdash_e^{fp} 2 : \textbf{real}}\ (50)}{\pi_{\mathbf{P}} \vdash_e^{fp} 1 + 2 : \textbf{real}}\ (43)$$

4.3.11. Equivalencia de Tipos

Se puede observar que ciertos tipos de nuestro sistema son equivalentes entre sí, a pesar de utilizar construcciones sintácticas diferentes. Un ejemplo es la declaración de un sinónimo de tipo **syn** $tn = \theta$. Sintácticamente ambos elementos serán distintos, ya que el primero se representará solo con su nombre tn y sus argumentos (si tuviese), mientras que el segundo podrá ser un tipo θ válido cualquiera del lenguaje. En determinados contextos queremos que la igualdad sea válida.

Reglas para Equivalencia de Tipos

Durante el análisis de un programa si un tipo θ es equivalente a otro tipo θ', entonces en cualquier contexto en el que se espera una expresión e del primer tipo, se podrá utilizar una expresión e' del segundo tipo. El *juicio de equivalencia* determina que un tipo θ es equivalente a otro tipo θ', bajo el contexto de tipos $\pi_{\mathbf{T}}$.

$$\pi_{\mathbf{T}} \vdash_u \theta \sim \theta'$$

El siguiente conjunto de reglas determina que el concepto introducido forma una relación de equivalencia entre los tipos de nuestro lenguaje. La relación binaria $\sim$ satisface las propiedades de reflexividad, simetría, y transitividad.

Regla para Equivalencia de Tipos 51. Reflexividad

$$\frac{}{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta}$$

Regla para Equivalencia de Tipos 52. Simetría

$$\frac{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta'}{\pi_{\mathbf{T}} \vdash_u \theta' \sim \theta}$$

Regla para Equivalencia de Tipos 53. Transitividad

$$\frac{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta' \qquad \pi_{\mathbf{T}} \vdash_u \theta' \sim \theta''}{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta''}$$

Una de las reglas más interesantes es la que permite relacionar una equivalencia entre tipos con el *juicio de tipado* de una expresión. Dado el *juicio* para una expresión particular, con el uso de esta regla es posible derivar un nuevo *juicio* para la misma expresión donde se intercambia el tipo original por su tipo equivalente.

Regla para Equivalencia de Tipos 54. Equivalencia

$$\frac{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{fp} e : \theta \qquad \pi_{\mathbf{T}} \vdash_u \theta \sim \theta'}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{fp} e : \theta'}$$

Claramente es necesario definir las reglas de equivalencia restantes para las construcciones sintácticas de tipos del lenguaje. Comenzando con los punteros, la deducción es bastante simple. Si ciertos tipos cualquiera son equivalentes, entonces un puntero que referencia a uno de estos es equivalente a un puntero que señala al otro.

Regla para Equivalencia de Tipos 55. Punteros

$$\frac{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta'}{\pi_{\mathbf{T}} \vdash_u \textbf{pointer } \theta \sim \textbf{pointer } \theta'}$$

Para los arreglos, se aplica una idea idéntica a la anterior. Si el tipo de valores que almacena un arreglo es equivalente a otro tipo cualquiera, entonces el arreglo es equivalente a otra estructura cuyo tipo interno es este último tipo. Notar que las dimensiones de los arreglos deben coincidir.

Regla para Equivalencia de Tipos 56. Arreglos

$$\frac{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta'}{\pi_{\mathbf{T}} \vdash_u \textbf{array } as_1, \ldots, as_n \textbf{ of } \theta \sim \textbf{array } as_1, \ldots, as_n \textbf{ of } \theta'}$$

En el caso de los tipos definidos, la idea empleada es análoga a la utilizada en las reglas previas. Si cada uno de los parámetros de tipo es equivalente a cierto tipo particular, entonces el tipo definido es equivalente a su aplicación con los parámetros intercambiados.

Regla para Equivalencia de Tipos 57. Tipos Definidos

$$\frac{\pi_{\mathbf{T}} \vdash_u \theta_i \sim \theta'_i}{\pi_{\mathbf{T}} \vdash_u tn \textbf{ of } \theta_1, \ldots, \theta_n \sim tn \textbf{ of } \theta'_1, \ldots, \theta'_n}$$

Las reglas más relevantes de todo este contexto, son las que nos permiten probar la equivalencia de un sinónimo de tipo con el tipo que lo define. Las cuales permiten intercambiar con libertad el uso de un tipo declarado, con el tipo de su respectiva definición a lo largo de un programa. Para el caso puntual de los sinónimos sin parámetros, la derivación es directa ya que solo es necesario demostrar su existencia en el correspondiente contexto de sinónimos definidos.

Regla para Equivalencia de Tipos 58. Sinónimos sin Argumentos

$$\frac{(tn, \emptyset, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t \vdash_u tn \sim \theta}$$

En el caso de un sinónimo con parámetros, es necesario realizar la sustitución apropiada. Aplicando la sustitución de variables de tipo al cuerpo de su definición, de acuerdo a los parámetros declarados junto con los argumentos recibidos, se obtiene el tipo equivalente.

Regla para Equivalencia de Tipos 59. Sinónimos con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t \vdash_u tn \textbf{ of } \theta_1, \ldots, \theta_n \sim \theta \mid [tv_1 : \theta_1, \ldots, tv_n : \theta_n]_{tv}}$$

Ejemplo de Prueba

Estamos en condiciones para demostrar la igualdad entre la *lista* declarada, y el *puntero a nodo* de su definición. Siguiendo en el contexto de chequeo de tipos del procedimiento *tail*, continuaremos con la prueba del fragmento (Código 4.3). Notar que en el código se combinan expresiones de ambos tipos de forma intercambiable, por lo que la siguiente prueba es fundamental para comprobar la validez de las sentencias del procedimiento que aún restan por verificar.

Prueba 9. Derivación de equivalencia de *lista* y *puntero a nodo*.

$$\dfrac{\dfrac{\pi_{\mathbf{FP}}(tail) = (\{\mathbf{in/out}\ l : list\ \mathbf{of}\ T\}, \emptyset)}{\emptyset_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{tail} l : list\ \mathbf{of}\ T}\ (38) \qquad \dfrac{(list, \{A\}, \mathbf{pointer}\ node\ \mathbf{of}\ A) \in \pi_s}{\emptyset_e, \pi_s, \pi_t \vdash_u list\ \mathbf{of}\ T \sim \mathbf{pointer}\ node\ \mathbf{of}\ T}\ (59)}{\emptyset_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{tail} l : \mathbf{pointer}\ node\ \mathbf{of}\ T}\ (54)$$

Notar en la siguiente derivación que el tipo de la variable p es *puntero a nodo*, por lo que difiere del tipo del parámetro l el cual es *lista*. Al introducir las reglas sobre equivalencia de tipos, somos capaces de dar una derivación para esta asignación.

Prueba 10. Demostración de corrección para inicialización de variable.

$$\dfrac{\dfrac{(p, \theta) \in \pi_v}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_e^{tail} p : \theta}\ (36) \qquad \dfrac{\text{Prueba 9}}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_e^{tail} l : \theta}\ (54)}{\pi_{\mathbf{T}}, \pi_{\mathbf{FP}}, \emptyset_{sn}, \pi_v \vdash_s^{tail} p := l}\ (24)$$

donde $\theta = \mathbf{pointer}\ node\ \mathbf{of}\ T$

Para finalizar la derivación completa del procedimiento *tail*, solo resta probar la sentencia que asigna al sucesor del primer elemento como nuevo primer elemento de la lista. La primer derivación involucra a los operadores para el acceso de variables, los cuales comprenden al de punteros y al de tuplas. Notar que se aplicó la traducción adecuada al *azúcar sintáctico* empleado en el código. La segunda consiste propiamente de la asignación de las listas involucradas.

Prueba 11. Demostración de corrección para acceso a variable.

$$\dfrac{\dfrac{\dfrac{\text{Prueba 9}}{\emptyset_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{tail} l : \theta}\ (54)}{\emptyset_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{tail} \star l : node\ \mathbf{of}\ T}\ (40) \qquad (node, \{Z\}, \{elem : \theta_1, next : \theta_2\}) \in \pi_t}{\emptyset_e, \pi_s, \pi_t, \pi_{\mathbf{FP}}, \pi_{\mathbf{V}} \vdash_e^{tail} \star l.next : \theta}\ (41)$$

donde $\theta = \mathbf{pointer}\ node\ \mathbf{of}\ T$, y los campos del *nodo* son $\theta_1 = Z$, y $\theta_2 = \mathbf{pointer}\ node\ \mathbf{of}\ Z$.

Prueba 12. Demostración de corrección para asignación de *lista*.

$$\dfrac{\dfrac{\text{Prueba 9}}{\pi_{\mathbf{P}} \vdash_e^{tail} l : \theta}\ (54) \qquad \dfrac{\text{Prueba 11}}{\pi_{\mathbf{P}} \vdash_e^{tail} \star l.next : \theta}\ (41)}{\pi_{\mathbf{P}} \vdash_s^{tail} l := \star l.next}\ (24)$$

donde $\theta = \mathbf{pointer}\ node\ \mathbf{of}\ T$

4.3.12. Chequeos para Programas

Finalmente para concluir con las reglas de derivación de los *juicios de tipado*, solo resta formalizar el juicio y la regla para el chequeo de un programa. El siguiente *juicio de tipado* determina que el chequeo de tipos de un programa es válido bajo los contextos $\pi_{\mathbf{T}}$ y $\pi_{\mathbf{FP}}$.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{FP}} \vdash_p td_1 \dots td_n \quad fpd_1 \dots fpd_m$$

La única regla asociada a este *juicio* tiene como premisas a las derivaciones de juicios para las declaraciones de tipo y las declaraciones de funciones y procedimientos, que conforman al programa. Notar que los contextos involucrados en el análisis se construyen de manera incremental, lo cual manifiesta la relevancia del orden de las declaraciones que aparecen en el programa.

Regla para Programas 60. Programas

$$\frac{\pi_{\mathbf{T}}^{i-1} \vdash_{td} td_i : \pi_{\mathbf{T}}^{i} \qquad \pi_{\mathbf{T}}^{n}, \pi_{\mathbf{FP}}^{j-1} \vdash_{fp} fpd_j : \pi_{\mathbf{FP}}^{j}}{\pi_{\mathbf{T}}^{0}, \pi_{\mathbf{FP}}^{0} \vdash_{p} td_1 \ldots td_n \quad fpd_1 \ldots fpd_m}$$

4.4. Más Chequeos Estáticos

Los *juicios de tipado* junto con sus reglas, conforman una parte importante de los chequeos estáticos. En esta sección definiremos formalmente otros chequeos estáticos, los cuales pueden ser pensados como predicados sobre programas que aseguran la corrección de más propiedades sobre estos. En una función, quisiéramos evitar que en el cuerpo se asignen los parámetros de su prototipo, mientras que para el caso del retorno es esperable que ocurra. En un procedimiento, los parámetros denotados como lectura no deben ser asignados, y viceversa, los parámetros denotados como escritura no pueden ser utilizados como valores.

4.4.1. Funciones para Variables

Para formalizar estas propiedades sobre funciones y procedimientos del programa, definiremos una serie de funciones auxiliares adicionales. Su utilidad radicará en que permiten distinguir el uso que se hace de los identificadores de variables en el cuerpo de las funciones, y los procedimientos.

Variables Libres

En el cálculo lambda, se denomina *libre* a toda variable que no se encuentra cuantificada. En nuestro caso, el lenguaje no provee ninguna especie de cuantificación a nivel de expresiones, por lo que toda variable en una expresión es una *variable libre*. La siguiente función calcula el conjunto de *variables libres* que ocurren en una expresión determinada. Notar que es posible aplicar un razonamiento análogo para calcular las ocurrencias *libres* en las sentencias del lenguaje, pero no nos será necesario.

$$FV^{\langle expression \rangle} : \langle expression \rangle \to \{\langle id \rangle\}$$

Notar que ciertos identificadores, como los nombres de funciones, no serán considerados por la función ya que estos no formarán parte de los chequeos que definiremos. Por cuestiones de claridad, los supraíndices de las funciones definidas serán omitidos.

$$\begin{aligned}
FV(ct) &= \emptyset \\
FV(f(e_1, \ldots, e_n)) &= FV(e_1) \cup \ldots \cup FV(e_n) \\
FV(e_1 \oplus e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\ominus e) &= FV(e) \\
FV(x) &= \{x\} \\
FV(v[e_1, \ldots, e_n]) &= FV(v) \cup FV(e_1) \cup \ldots \cup FV(e_n) \\
FV(v.fn) &= FV(v) \\
FV(\star v) &= FV(v)
\end{aligned}$$

Variables de Escritura

Una variable es considerada de *escritura*, cuando su valor será potencialmente modificado durante la ejecución de una sentencia. Las siguientes funciones calculan el conjunto de *variables de escritura* que ocurren en expresiones o sentencias, respectivamente. Debido a que necesitamos información sobre las etiquetas declaradas para cada parámetro introducido en el prototipo de los procedimientos, tenemos que indexar la definición de la función en el contexto que contiene a las declaraciones de procedimientos. Notar que esta propiedad a definir (junto con las que vendrán) asume que el programa está bien tipado; es decir, podemos construir una derivación para un determinado juicio de tipado (y por lo tanto, construir el contexto que necesitamos).

$$WV^{\langle expression\rangle} : \langle expression\rangle \to \{\langle id\rangle\} \qquad WV^{\langle sentence\rangle}_{\pi_p} : \langle sentence\rangle \to \{\langle id\rangle\}$$

Las sentencias que pueden modificar el estado de una variable son la asignación, el procedimiento nativo *alloc*, y la llamada de procedimientos. Notar el abuso de notación al aplicar la función a un bloque de sentencias ss, en esta situación el conjunto de *variables de escritura* se obtiene al acumular la aplicación sobre cada una de las sentencias que lo conforman. Supongamos que la condición $(p, \{io_1\ a_1 : \theta_1, \ldots, io_n\ a_n : \theta_n\}, cs) \in \pi_p$ es válida.

$$\begin{array}{lcl}
WV_{\pi_p}(\textbf{skip}) & = & \emptyset \\
WV_{\pi_p}(v := e) & = & WV(v) \\
WV_{\pi_p}(\textbf{alloc}\ v) & = & WV(v) \\
WV_{\pi_p}(\textbf{free}\ v) & = & \emptyset \\
WV_{\pi_p}(\textbf{if}\ e\ \textbf{then}\ ss\ \textbf{else}\ ss') & = & WV_{\pi_p}(ss) \cup WV_{\pi_p}(ss') \\
WV_{\pi_p}(\textbf{while}\ e\ \textbf{do}\ ss) & = & WV_{\pi_p}(ss) \\
WV_{\pi_p}(\textbf{for}\ x := e_1\ \textbf{to}\ e_2\ \textbf{do}\ ss) & = & WV_{\pi_p}(ss) - \{x\} \\
WV_{\pi_p}(\textbf{for}\ x := e_1\ \textbf{downto}\ e_2\ \textbf{do}\ ss) & = & WV_{\pi_p}(ss) - \{x\} \\
WV_{\pi_p}(p(e_1, \ldots, e_n)) & = & \{WV(e_i) \mid io_i = \textbf{out} \vee io_i = \textbf{in/out}\}
\end{array}$$

La definición para el caso de la sentencia *for* es interesante; la variable x es declarada y actualizada de forma implícita dentro del cuerpo de la sentencia. Por lo que esta variable quedará fuera del conjunto de variables que deseamos considerar como asignables.

Otro caso interesante de la definición es la llamada a procedimientos, ya que se aplica a una serie de expresiones que podrían llegar a ser modificadas en caso de corresponderse con un parámetro del procedimiento etiquetado con **out** o **in/out**. En cambio, el resto de los argumentos que correspondan a un parámetro de entrada no serán modificados por la llamada.

Técnicamente ninguna ocurrencia de una variable en una expresión es de *escritura*, ya que su evaluación nunca debería producir efectos secundarios. En virtud que una expresión puede suceder dentro de una sentencia, la cual tiene el potencial de modificar el estado de las variables que ocurren en ella, la función tendrá el siguiente comportamiento. Notar que en soledad la definición no tiene utilidad, solo posee sentido al complementar la función para sentencias.

$$\begin{array}{lcll}
WV(e) & = & \emptyset & e \notin \langle variable\rangle \\
WV(x) & = & \{x\} & \\
WV(v[e_1, \ldots, e_n]) & = & WV(v) & \\
WV(v.fn) & = & WV(v) & \\
WV(\star v) & = & \emptyset &
\end{array}$$

Es claro que no es posible modificar el valor de una expresión, la cual no representa a una variable, por lo que el comportamiento de la función resulta trivial en este aspecto. Para los operadores de variables, en el acceso de arreglos o tuplas, se debe propagar la función hasta obtener el identificador de la variable inicial. En el caso de los punteros, hay un detalle particular que es importante aclarar sobre el *aliasing*.

El *aliasing* es una situación que ocurre cuando la misma posición de memoria se puede acceder utilizando distintos identificadores. En nuestro caso, se da cuando hay más de un puntero diferente que referencia a la misma estructura en memoria. Debido a esta situación, resulta complejo analizar estáticamente cuando el elemento referido por un puntero podrá ser potencialmente modificado. Por lo tanto, esta condición implica que la definición de la propiedad que pretendemos dar probablemente no sea completa.

Variables de Lectura

Una variable es considerada de *lectura* cuando su valor es potencialmente utilizado en la evaluación de una expresión durante la ejecución de una sentencia. Al igual que ocurría con el cálculo de las *variables de escritura*, acá también es necesario asumir el correcto tipado del programa; particularmente de sus procedimientos.

$$RV^{\langle expression\rangle} : \langle expression\rangle \to \{\langle id\rangle\} \qquad RV_{\pi_p}^{\langle sentence\rangle} : \langle sentence\rangle \to \{\langle id\rangle\}$$

Es interesante notar que el conjunto de *variables de lectura* de una sentencia, no necesariamente será disjunto al correspondiente conjunto de *variables de escritura*. Un ejemplo concreto podría ser la asignación de una variable a si misma $x := x$, donde la primer ocurrencia es de *escritura* y la segunda ocurrencia es de *lectura*.

La siguiente declaración no será una definición dual a la función previa, principalmente por dos razones. La primera es que cualquier *variable libre* que ocurra en un sentencia, podrá ser considera de lectura siempre y cuando no aparezca exclusivamente en la componente izquierda de una asignación. La segunda es que, inclusive en ese caso, podría ocurrir que exista una variable de lectura, en el lado izquierdo de la asignación, cuando es utilizada como un valor para referirse a alguna componente de una estructura más compleja. Por ejemplo, el acceso a una posición de memoria. Supongamos que la condición $(p, \{io_1\ a_1 : \theta_1, \ldots, io_n\ a_n : \theta_n\}, cs) \in \pi_p$ es válida.

$$\begin{aligned}
&RV_{\pi_p}(\textbf{skip}) &&= \emptyset\\
&RV_{\pi_p}(v := e) &&= RV(v) \cup FV(e)\\
&RV_{\pi_p}(\textbf{alloc}\ v) &&= RV(v)\\
&RV_{\pi_p}(\textbf{free}\ v) &&= FV(v)\\
&RV_{\pi_p}(\textbf{if}\ e\ \textbf{then}\ ss\ \textbf{else}\ ss') &&= FV(e) \cup RV_{\pi_p}(ss) \cup RV_{\pi_p}(ss')\\
&RV_{\pi_p}(\textbf{while}\ e\ \textbf{do}\ ss) &&= FV(e) \cup RV_{\pi_p}(ss)\\
&RV_{\pi_p}(\textbf{for}\ x := e_1\ \textbf{to}\ e_2\ \textbf{do}\ ss) &&= FV(e_1) \cup FV(e_2) \cup (RV_{\pi_p}(ss) - \{x\})\\
&RV_{\pi_p}(\textbf{for}\ x := e_1\ \textbf{downto}\ e_2\ \textbf{do}\ ss) &&= FV(e_1) \cup FV(e_2) \cup (RV_{\pi_p}(ss) - \{x\})\\
&RV_{\pi_p}(p(e_1, \ldots, e_n)) &&= \{FV(e_i) \mid io_i = \textbf{in} \vee io_i = \textbf{in/out}\} \cup \ldots\\
&&&\quad \ldots \cup \{RV(e_i) \mid io_i = \textbf{out}\}
\end{aligned}$$

De forma similar a la definición previa para el caso de la llamada a un procedimiento, si una expresión corresponde a un parámetro de entrada, entonces todas las ocurrencias de variables serán de *lectura*. Adicionalmente podría ocurrir una situación similar a la que anteriormente mencionamos con la asignación; una expresión que es utilizada como argumento para un parámetro que fue etiquetado con **out**, podría contener *variables de lectura*.

Claramente toda ocurrencia de variable en una expresión es de *lectura*, ya que su valor será potencialmente utilizado en la evaluación de la misma. Sin embargo la siguiente definición complementa la definición para sentencias, por lo que de forma aislada puede resultar contra intuitiva. En este caso es posible pensar a la función para *variables de lectura* como una versión dual de la función para *variables de escritura*.

$$\begin{array}{lll} RV(e) & = FV(e) & e \notin \langle variable \rangle \\ RV(x) & = \emptyset & \\ RV(v[e_1, \ldots, e_n]) & = RV(v) \cup FV(e_1) \cup \ldots \cup FV(e_n) & \\ RV(v.fn) & = RV(v) & \\ RV(\star v) & = FV(v) & \end{array}$$

Cuando una expresión no es una variable entonces solo podrá ser evaluada, lo cual implica que todas sus ocurrencias de variables serán de *lectura*. Para los operadores de variables, solamente se consideran a las expresiones empleadas para acceder a los diferentes elementos que componen a la estructura de la variable. En el caso de los punteros, todas las variables que intervienen en el acceso a memoria son consideradas de *lectura*.

4.4.2. Predicados sobre Funciones y Procedimientos

Diremos que un programa es válido estáticamente cuando existe una derivación para su *juicio de tipado*, y se satisfacen las propiedades sobre *lectura/escritura* de variables para todas sus declaraciones de funciones y procedimientos. Recordar que los contextos resultados del chequeo son importantes para la verificación de las propiedades.

Extenderemos las funciones previamente definidas para que puedan ser aplicadas al cuerpo de una función o de un procedimiento. De esta manera se logra flexibilizar la notación, lo cual facilitará la lectura de los predicados. La aplicación de las funciones debe ser propagada solamente a las sentencias que conforman al cuerpo, acumulando los resultados obtenidos.

$$WV_{\pi_p}(vd_1 \ldots vd_n\ s_1 \ldots s_m) = WV_{\pi_p}(s_1) \cup \ldots \cup WV_{\pi_p}(s_m)$$
$$RV_{\pi_p}(vd_1 \ldots vd_n\ s_1 \ldots s_m) = RV_{\pi_p}(s_1) \cup \ldots \cup RV(s_m)$$

Predicados para Funciones

Supongamos tenemos la siguiente definición de función.

$$\textbf{fun}\ f\ (a_1 : \theta_1, \ldots, a_l : \theta_l)\ \textbf{ret}\ a_r : \theta_r\ \textbf{where}\ cs\ \textbf{in}\ body$$

Puesto que una función siempre devuelve un valor, no sería posible retornar algún resultado si la variable correspondiente no fuese asignada. La variable de retorno a_r debe ser modificada por las sentencias de la función. Hay que tener en cuenta que estáticamente no es posible verificar si todas las posiciones de memoria alcanzables por la variable fueron debidamente inicializadas en el cuerpo. Lo cual implica que la verificación no es completa, en el sentido que podría ser satisfecha y de todas maneras el retorno no ser asignado. Además es necesario verificar que ninguno de los argumentos a_i sea modificado por las sentencias que conforman al cuerpo.

Predicado sobre Lectura/Escritura 1. Retorno de Función

$$a_r \in WV_{\pi_p}(body)$$

Predicado sobre Lectura/Escritura 2. Argumento de Función

$$\{a_1, \ldots, a_l\} \cap WV_{\pi_p}(body) = \emptyset$$

Predicados para Procedimientos

Supongamos tenemos la siguiente definición de procedimiento.

$$\textbf{proc}\ p\ (io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l)\ \textbf{where}\ cs\ \textbf{in}\ body$$

Es necesario asegurar el correcto uso de los parámetros del procedimiento, de acuerdo a la etiqueta de *entrada/salida* con la que fueron especificados. Un parámetro definido como **in**, solo podrá ser utilizado como una variable de lectura. Un parámetro definido como **out**, solo podrá ser utilizado como una variable de escritura. Por supuesto, un parámetro declarado como **in/out**, podrá ser utilizado como una variable que cumple ambos roles y no es necesario realizar ninguna verificación al respecto.

Predicado sobre Lectura/Escritura 3. Entrada de Procedimiento

$$\{a_i \mid io_i = \textbf{in}\} \cap WV_{\pi_p}(body) = \emptyset$$

Predicado sobre Lectura/Escritura 4. Salida de Procedimiento

$$\{a_i \mid io_i = \textbf{out}\} \cap RV_{\pi_p}(body) = \emptyset$$

Ejemplo de Predicado

Los predicados definidos pueden ser aplicados a las funciones y los procedimientos que fueron presentados a lo largo de este trabajo. Para varios de estos ejemplos ya probamos que están bien tipados, por lo tanto nos limitaremos a calcular sus conjuntos de *variables de lectura* y *variables de escritura*. Para facilitar la comprensión de los ejemplos, utilizaremos la notación $body_{fp}$ para representar al cuerpo de la función o el procedimiento fp.

Comenzando con la implementación del algoritmo de ordenación, observar que el arreglo a ordenar a es empleado como una variable de *lectura* y *escritura* por ambos procedimientos. En el fragmento (Código 4.4), se emplean los parámetros de entrada i y j de manera apropiada. En el código (Código 4.5), ninguna de las variables declaradas por las sentencias *for* es considerada en los conjuntos obtenidos.

$$WV_{\pi_p}(body_{swap}) = \{temp, a\} \qquad WV_{\pi_p}(body_{selectionSort}) = \{minP, a\}$$
$$RV_{\pi_p}(body_{swap}) = \{temp, a, i, j\} \qquad RV_{\pi_p}(body_{selectionSort}) = \{minP, a\}$$

Pasando a las operaciones sobre *listas abstractas*, el análisis es directo. En el fragmento (Código 4.2), el retorno b es una variable de *escritura*, mientras que el argumento l es una variable de *lectura*. En el código (Código 4.3), el parámetro l cumple ambos roles, y se puede notar que la variable p declarada en el cuerpo solo es utilizada para su inicialización y subsiguiente liberación.

$$WV_{\pi_p}(body_{isEmpy}) = \{b\} \qquad WV_{\pi_p}(body_{tail}) = \{l, p\}$$
$$RV_{\pi_p}(body_{isEmpty}) = \{l\} \qquad RV_{\pi_p}(body_{tail}) = \{l\}$$

4.5. Implementación de Chequeos Estáticos

Para terminar con el capítulo presentaremos los detalles principales sobre un primer prototipo de implementación para los chequeos estáticos. Decimos prototipo porque, a diferencia de la implementación del parser, todavía quedan muchos aspectos por desarrollar y mejorar.

4.5.1. Diseño General

Estudiando el desarrollo de *Castegren* y *Reyes* [7], se adaptó el diseño presentado en este trabajo para nuestra implementación de los chequeos estáticos. El artículo reporta las experiencias durante el desarrollo de un compilador para un lenguaje concurrente orientado a los objetos, utilizando para su implementación el lenguaje funcional *Haskell*. El foco del trabajo consiste en la implementación del *chequeo de tipos*, donde de manera incremental se agregan funcionalidades complejas adicionales al diseño general inicial. Se hace fuerte hincapié en el uso de mónadas, como una herramienta fundamental para el desarrollo de los chequeos estáticos.

La mónada `RWS` es empleada para definir los *chequeos estáticos* (Código 4.6) del lenguaje. El entorno de lectura `Scope`, indica el identificador de la función o el procedimiento que actualmente está siendo verificado. La salida de escritura `SCWarnings`, consiste del listado de advertencias generadas durante el análisis estático. El estado variable `Context`, comprende toda la información reunida durante la verificación del programa; su composición es similar a los contextos formalizados durante los *juicios de tipado*. Al encontrar un error estático `SCError`, se utiliza la mónada `Except` para abortar el análisis, generando el mensaje de error adecuado.

```
1 -- StaticCheck Monad
2 type Check = RWST Scope SCWarnings Context (Except SCError)
```

Código 4.6: Mónada para Chequeos Estáticos

Es fundamental que durante el análisis, cuando se genera un error o una advertencia, sea posible indicar *donde* ocurre cada una. Se define a la *traza* del programa (Código 4.7) como una lista de *nodos*, donde cada uno posee información específica del elemento sintáctico que representa. La definición del tipo `Backtrace` nos permite llevar un contexto global sobre donde ocurre el chequeo estático.

La traza es actualizada cuando el análisis visita un nuevo nodo, lo cual permite que la posición sea rastreada de manera apropiada. Durante la verificación, su comportamiento es similar al de una pila. Cuando se analiza un nuevo elemento sintáctico se genera un nodo con la información correspondiente, para usar en el caso de necesitar producir un mensaje de error o advertencia. Si no hay ningún problema, el *chequeador* elimina al nodo creado y retorna la traza al estado anterior, continuando con el análisis del programa.

```
1 -- Backtrace
2 newtype Backtrace = BT [BTNode]
```

Código 4.7: Traza de Análisis

Cuando la verificación del programa falla, se genera un error estático (Código 4.8) con un mensaje descriptivo sobre su causa, junto con la respectiva traza de análisis y la posición en el archivo donde ocurre. El *chequeador estático* solo reporta un único error a la vez, pero es posible extenderlo para soportar múltiples mensajes de error. En el artículo se describe una manera factible para la implementación de esta característica.

```
1 -- StaticCheck Error
2 data SCError = SCE { errorMsg :: Error
3                    , errorBT  :: Backtrace
4                    }
```

Código 4.8: Error en Análisis Estático

El *chequeador estático* acumulará advertencias (Código 4.9) durante el análisis, y en caso de no encontrar errores, deberá retornarlas una vez finalizado. Una advertencia consiste de un

mensaje descriptivo, junto con la traza de análisis adecuada y la posición en el archivo donde ocurre.

```
-- StaticCheck Warning
data SCWarning = SCW { warningMsg :: Warning
                     , warningBT  :: Backtrace
                     }

type SCWarnings = [SCWarning]
```

Código 4.9: Advertencia en Análisis Estático

Durante el análisis del programa, la información reunida es almacenada en el contexto general (Código 4.10). Similar a la teoría, es necesario acumular las construcciones declaradas junto con los elementos introducidos en el alcance actual. El contexto general está compuesto por la *traza* de análisis, y una serie de contextos específicos; como el de tipos definidos `CType`, el de funciones y procedimientos declarados `CFunProc`, y el de variables de tipo, identificadores para tamaños de arreglos, y variables introducidas en el respectivo prototipo o cuerpo siendo analizado `CBlock`. Cada uno de los *datatypes* mencionados, cuenta con un conjunto de operaciones propias para trabajar con los datos almacenados en sus estructuras.

```
-- Context
data Context = C { bTrace    :: Backtrace
                 , cFunProc  :: CFP.CFunProc
                 , cType     :: CT.CType
                 , cBlock    :: CB.CBlock
                 }
```

Código 4.10: Contexto de Análisis

Ciertas reglas para la derivación de los *juicios de tipado*, son indexadas por el nombre de la función o el procedimiento siendo analizado. Esta particularidad es adaptada en la implementación (Código 4.11). Es importante mencionar que si se consultara la *traza* de análisis presente, sería posible obtener el identificador correspondiente a la construcción actual; evitando de esta forma la declaración del *datatype*. Por cuestiones de simplicidad en la implementación, se prefirió conservar la definición mencionada.

```
-- Scope
data Scope = InFun  {getID :: Id}
           | InProc {getID :: Id}
           | InType {getTN :: TName}
           | Out
```

Código 4.11: Alcance de Análisis

4.5.2. Módulos

En virtud de la complejidad para el desarrollo del *chequeador estático*, se decidió separar su implementación en varios directorios. Cada uno representa conceptos propios, y se encarga de ciertos aspectos específicos sobre la verificación del programa. A continuación, describiremos los detalles más relevantes sobre los módulos que componen esta implementación.

Error

El directorio *Error* está conformado por tres módulos diferentes. La *traza* de análisis y sus operadores propios, junto con los *nodos* que almacenan la información relevante sobre la posición del *chequeador estático*, se encuentran definidos en *Error.Backtrace*. Haciendo uso de las construcciones declaradas en este módulo, se especifican los errores y las advertencias sobre los *chequeos estáticos*. Los errores estáticos, junto con los respectivos mensajes informativos de error, son definidos en *Error.Error*. Las advertencias estáticas, junto con los respectivos mensajes informativos de advertencia, son definidas en *Error.Warning*. La organización de los módulos fue adaptada de la bibliografía previamente citada.

Monad

El directorio *Monad* está conformado por cinco módulos diferentes. La declaración de la mónada para los chequeos estáticos se encuentra en el módulo principal, *Monad.Monad*. La totalidad de las operaciones sobre la mónada está propiamente definida en el módulo. Se especifican las funciones para generar errores `errorSC` y advertencias `warningSC`, al igual que para administrar la *traza* de análisis `withBT`. Adicionalmente todas las operaciones específicas sobre el contexto general y el alcance de análisis, son definidas de manera adecuada para la mónada.

La información contextual necesaria para efectuar el análisis del programa, se divide en los contextos específicos mencionados anteriormente. Cada uno de los contextos es declarado en un módulo particular, junto con todos los operadores esenciales para acceder a su información almacenada. En *Monad.FunProc*, se define el contexto para funciones y procedimientos. En *Monad.Type*, se define el contexto para tipos de datos. En *Monad.Block*, se define el contexto para variables de tipo, identificadores para tamaños de arreglos, y variables. Notar que en el último contexto, también se almacenan las restricciones de clases que aún no han sido empleados en el respectivo cuerpo de la función o el procedimiento; la información es fundamental para el caso que fuese necesario generar una advertencia sobre la especificación de restricciones sin uso. Por último, el *alcance* de análisis que determina si el *chequeo de tipos* se encuentra dentro de una función, un procedimiento, o una declaración de tipo, se define en *Monad.Scope*.

Check

El directorio *Check* está conformado por ocho módulos diferentes, en los cuales se encuentra la implementación de los *juicios de tipado* y los demás *chequeos estáticos* definidos formalmente en este capítulo.

La llamada principal para la verificación de un programa se realiza en *Check.Program*. Su análisis se divide de acuerdo a los tipos declarados, y las funciones y los procedimientos definidos. Primero se efectúa el análisis para la declaración de tipos de datos, donde la verificación se especifica en *Check.TypeDecl*; en tanto que la validación para los tipos del lenguaje se detalla en *Check.Type*. Segundo se efectúa el análisis para las funciones y los procedimientos definidos, donde su verificación se especifica en el módulo *Check.FPDecl*; entre tanto los chequeos generales para sus respectivos cuerpos, y en particular las declaraciones de variables, se detallan en *Check.Block*. La definición para las funciones sobre la escritura y lectura de variables, al igual que las verificaciones asociadas al uso de variables, son especificadas en *Check.Variable*. Por último, el análisis de las sentencias del lenguaje se realiza en *Check.Statement*; a la vez que el análisis para las expresiones se detalla en *Check.Expr*.

TypeSystem

El directorio `TypeSystem` está conformado por cuatro módulos diferentes. La implementación en cuestión es la responsable que esta primer versión del lenguaje sea solo un prototipo. Para resolver el polimorfismo que admiten ciertas construcciones del lenguaje, la implementación resulta ser una solución *ad hoc* básica.

En `TypeSystem.TypeSystem` se define un sistema de tipos de *uso interno* para nuestro *chequeador de tipos*. En este módulo se implementan, por ejemplo, a los tipos enumerados o a las estructuras de tipo tupla como tipos del sistema, y en el caso de los sinónimos, son reemplazados por el tipo al que representan. Además se implementa un tipo polimórfico que es definido específicamente para la constante **null**.

En `TypeSystem.Instance` se definen las instancias de clases satisfechas por cada uno de los tipos del sistema. Notar que las propiedades descriptas previamente en la documentación, son adaptadas en la implementación. Incluso también se declaran las instancias para los tipos enumerables, a pesar de no formar parte de una clase formalmente definida.

En `TypeSystem.Transform` se definen las operaciones para convertir un tipo de la sintaxis del lenguaje, a un tipo del sistema de tipos. La transformación es directa, salvo para los tipos definidos donde es necesario realizar un paso adicional para separarlos de acuerdo a su naturaleza. En el caso de los sinónimos, se aplica la transformación para obtener al tipo que representan; de cierta manera, no existen los sinónimos de tipo en el sistema. Adicionalmente, se especifican las funciones auxiliares para obtener el tipo sintáctico de la variable declarada por la sentencia *for*.

En `TypeSystem.Unification` se define un algoritmo de *unificación* bien primitivo. En la llamada de una función o un procedimiento, es necesario demostrar la existencia de las funciones de sustitución adecuadas para igualar el tipo esperado por los parámetros, con el tipo recibido por los argumentos. En el caso de la implementación, con `unification` se intentan construir las sustituciones mencionadas de manera incremental. Iniciando con un *contexto de unificación* vacío, se procura incorporar las sustituciones pertinentes a variables de tipo e identificadores de tamaños, para hacer coincidir los tipos esperados con los tipos recibidos. En caso exitoso, se verifica la satisfacción de las restricciones de clase de acuerdo a la sustitución construida.

Capítulo 5

Conclusión

En este trabajo final hemos definido formalmente e implementado un nuevo lenguaje. Se han descripto las motivaciones principales que justificaron su creación, y se hizo mención de los elementos más relevantes que lo definen. Desde un punto de vista formal, las contribuciones de la tesis se pueden resumir en:

- La definición informal de las características deseables para una primera versión del lenguaje. Se estudiaron de manera exhaustiva los ejemplos del pseudocódigo de la materia, donde se hizo un resumen con todas las construcciones empleadas en el mismo, con lo que se obtuvo una primera noción de un lenguaje con muchas características interesantes. Algunas de las cuales son el polimorfismo paramétrico y *ad hoc*, la posibilidad de emplear distintos tipos de valores para delimitar los índices válidos de arreglos, la capacidad de declarar instancias de clases, y la definición de estructuras iterables.

- La definición de su sintaxis, abstracta y concreta, lo cual permite establecer de manera precisa la forma de escribir programas en el lenguaje. A través de las gramáticas correspondientes, logramos eliminar las ambigüedades presentes en determinadas estructuras sintácticas del pseudocódigo.

- La definición de los chequeos estáticos, que determinan cuando un programa está bien tipado antes de su ejecución. Para el caso del chequeo de tipos, se definen varias clases de *juicios de tipado* con sus respectivas reglas. Los chequeos no solo se limitan a verificaciones de tipo. También se analiza el uso adecuado de variables donde se evita la lectura o escritura de variables que no soportan las respectivas acciones.

Sumado al desarrollo matemático, realizamos la primer iteración en la implementación del lenguaje; donde utilizando la formalización como base, se han implementado las correspondientes etapas de *parser* y *chequeos estáticos*. Respecto a la implementación, las contribuciones del trabajo son:

- La implementación de las fases de *análisis léxico* y *análisis sintáctico*. Los principales desafíos fueron adecuar la librería *Parsec* a las necesidades de la implementación, y la posterior transición a la librería *Megaparsec*. Se estudiaron distintas opciones para la implementación de la sintaxis, la cual finalmente se adaptó al tipo de datos `Located`; permitiendo de esta manera almacenar la información de posición de todas las construcciones sintácticas relevantes, y facilitar la generación de mensajes de error precisos.

- La implementación de los chequeos estáticos, que comprenden las verificaciones estáticas que se realizan previa la ejecución de un programa. Utilizando como guía el trabajo [7], y la formalización de los *juicios de tipado*, se desarrolló una primera implementación incompleta del lenguaje. La transición del formalismo a la implementación fue casi directa, salvo por las dificultades encontradas en el algoritmo de *unificación* para inferir tipos, y el subtipado para los tipos numéricos.

5.1. Trabajos Futuros

El diseño del lenguaje aún está lejos de estar terminado, ya que este trabajo solo comprendió la primer etapa de su desarrollo. Una importante tarea a realizar será definir matemáticamente la semántica dinámica que nos permita ejecutar programas; probablemente utilizando una semántica operacional *small step* para poder ejecutar paso a paso un programa.

5.1.1. Enriquecer el Lenguaje

Existen una serie de características que fueron consideradas a lo largo del desarrollo de este trabajo, pero que no llegaron a ser incluidas en esta primera versión del lenguaje. A continuación daremos una breve descripción de cada una de estas, junto con sugerencias para facilitar su futura incorporación.

Soporte para Múltiples Módulos

Nuestro lenguaje solo permite definir programas en un único contexto. Lo cual resulta un inconveniente para su incorporación en el dictado de la materia. Debido que durante la asignatura se hace hincapié en la importancia de la separación de los módulos para la especificación de *tipos abstractos de datos*, y los correspondientes a su implementación, es fundamental que el lenguaje provea las herramientas necesarias para mantener esa abstracción mediante el encapsulamiento de las construcciones involucradas.

Para incorporar esta funcionalidad, se deberá determinar la sintaxis adecuada para poder importar y exportar tipos, funciones, y procedimientos definidos dentro de un determinado módulo. Sumado a esto, se tendrán que adaptar los chequeos estáticos para soportar estas nuevas situaciones. En particular los sinónimos de tipos actualmente son interpretados como una definición transparente, donde idealmente quisiéramos que su definición se vuelva opaca fuera del módulo en el que fueron declarados. Respecto a la implementación, se deberá adaptar la información de posición `Info` para almacenar además, el nombre del archivo `File` correspondiente a la ocurrencia del elemento sintáctico analizado.

Clases e Instancias

Actualmente en el lenguaje solo hay definidas dos clases, **Eq** y **Ord**. Las cuales representan a los tipos que permiten comparaciones entre sus valores, ya sea en base a su igualdad en el caso de la primera, o según su orden para la segunda. Una posible expansión del lenguaje sería el agregado de nuevas clases nativas. Por dar un ejemplo, una alternativa sería agregar la clase **Num**; la cual representaría a cualquier conjunto de valores que admita las operaciones de suma, resta, multiplicación, y demás operaciones numéricas.

Otra extensión del lenguaje es la posibilidad de definir instancias para los tipos declarados, donde ya existe una sintaxis pensada para estas construcciones. Actualmente solo algunos tipos

tienen definidas las instancias de **Eq** y **Ord** de forma nativa, donde no es posible definir nuevas instancias para los demás tipos.

```
type node = tuple
            value : int,
            next  : pointer of node
            end tuple

inst Eq ( n1, n2 : node ) ret equal : bool
  equal := n1.value == n2.value && n1.next == n2.next
end inst

end type
```

Código 5.1: Declaración de Instancia para Nodo

En el fragmento (Código 5.1) se ilustra un ejemplo particular, en el cual se declara la operación de comparación `==` para el tipo *node*. De esta manera sería posible definir una instancia para la clase `Eq` en el lenguaje. Inicialmente al limitarnos a solo dos clases, que caracterizan propiedades similares, la verificación de sus instancias se simplifica.

Estructuras Iterables

Durante el desarrollo del lenguaje se analizó la posibilidad de incluir de manera nativa la clase **Iter** para caracterizar a todos los tipos que poseen la capacidad de ser iterados, es decir que pueden ser *recorridos* de cierta forma obteniendo todos los elementos que los componen. Algunos tipos que podrían tener instancias de la clase serían los conjuntos y las listas, donde ambos representan una colección de elementos. Una vez que se define la instancia de la clase para un tipo en particular, se podrán utilizar valores del mismo dentro de la sentencia **for** x **in** e **do** $s_1 \dots s_m$, lo cual permite recorrerlos obteniendo uno por uno todos los elementos que lo conforman. El siguiente ejemplo calcula la sumatoria de elementos de un conjunto, suponiendo que el tipo `set` tiene instancia de **Iter**.

```
fun sumIter ( s : set of (int) ) ret sum : int
  sum := 0
  for i in s do
    sum := i + sum
  od
end fun
```

Código 5.2: Sumatoria de Conjunto Iterable

Es relevante aclarar que a pesar de referirnos a **Iter** como una clase, aún no se ha determinado si la misma será definida como tal en el lenguaje, donde es posible que durante el desarrollo del proyecto su naturaleza sea modificada por lo que su interpretación cambie de manera acorde.

5.1.2. Respecto a la Implementación

Existe ciertos aspectos de la implementación que no han sido desarrollados por completo, o más bien, podrían ser mejorados para ofrecer un lenguaje más sofisticado.

Generación de Múltiples Errores

Uno de los aspectos que se puede mejorar de la implementación del lenguaje, es la generación de múltiples errores tanto en la etapa del *parser* como en la de *chequeos estáticos*. Actualmente cuando se encuentra un error durante el parsing o la verificación de un programa se aborta por completo el análisis del mismo, generando un mensaje de error sobre la causa de la falla. Una característica deseable en el lenguaje, es poder capturar la mayor cantidad posible de errores encontrados en una determinada etapa, antes de fallar, y generar todos los mensajes necesarios correspondientes.

- **Parser** La librería *Megaparsec* ofrece un mecanismo para señalar múltiples errores en una sola corrida del parser. Un requisito para utilizar esta funcionalidad es que debe ser posible omitir la sección problemática de la cadena de caracteres que estamos intentando parsear, para resumir el parseo en una posición que se considere estable. Lo cual se consigue con el uso de la función `withRecovery`, que permite continuar el parsing luego de ocurrido un error.

- **Chequeos Estáticos** Una posibilidad para la generación de múltiples errores en esta etapa, está descripta en el artículo que se empleó como guía para la implementación de los chequeos estáticos [7]. Esta opción consiste en la implementación de un combinador, que es invocado cada vez que se deben verificar una serie de elementos sintácticos. La idea es poder combinar la lista de chequeos a realizar y, si todos tienen éxito, se devuelven sus resultados correspondientes. En caso contrario, se deberán acumular la totalidad de los errores producidos, y luego generar los mensajes informativos adecuados.

Coersiones y Subtipado

En el lenguaje hay dos tipos numéricos, los valores enteros y los valores reales. En ambos casos se encuentra definido un listado de operadores sobrecargados que pueden ser utilizados para operar con cualquiera de los valores previos. Una restricción de la implementación actual, es que ambos tipos son percibidos como valores completamente diferentes. Lo cual significa, por ejemplo, que la operación de sumar enteros con reales es detectada como un error de tipos. Esto imposibilita implementar ciertos algoritmos donde deseamos realizar operaciones donde se combinan ambos tipos de valores numéricos, como es el caso de querer calcular el promedio de un arreglo de números reales, el cual posee un identificador para su tamaño. El código presentado a continuación (Código 5.3) ilustra la situación mencionada. Notar que al chequear la última asignación de la función, el lenguaje fallará debido a un error de tipos producido por la división de un número real por uno de tipo entero.

```
1 fun averageReal ( a : array [n] of real ) ret avg : real
2   var sum : real
3   sum := 0.0
4   for i := 1 to n do
5     sum := a[i] + sum
6   od
7   avg := sum / n {- Type Error -}
8 end fun
```

Código 5.3: Promedio en Arreglo de Reales

Inferencia de Tipos

El sistema de tipos que se encuentra actualmente implementado puede resultar básico a la hora de resolver ciertas cuestiones debido que solo es un primer prototipo. En consecuencia a esto se presentan algunas limitaciones, donde el algoritmo de unificación no es lo suficientemente sofisticado para resolver el polimorfismo admitido por funciones y procedimientos. Un ejemplo particular es la llamada de las construcciones mencionadas con la constante **null**. Tomando como ejemplo la implementación de la función (Código 5.4); la cual obtiene el elemento señalado por un puntero si el mismo no es nulo, o devuelve su segundo argumento en caso contrario. Si aplicamos la función con la llamada `pointerAccess(null,5)`, entonces se producirá un error en la etapa de chequeos estáticos del lenguaje, debido a la imposibilidad de chequear los tipos involucrados.

```
1 fun pointerAccess ( p : pointer of T, t : T) ret value : T
2   if p == null then
3     value := t
4   else
5     value := #p
6   fi
7 end fun
```

Código 5.4: Ejemplo para Inferencia de Tipos

El conflicto sucede cuando es necesario inferir un tipo para la constante **null**, combinada con el polimorfismo que admite una función o un procedimiento, debido que nuestro algoritmo de unificación no es lo suficientemente sofisticado para asignarle correctamente un tipo.

Para resolver este limitante, se presentan dos opciones que requieren la modificación de la implementación actual. La primera, orientada al tipado explícito, implica la adaptación de la sintaxis del lenguaje para forzar al programador a anotar el tipo concreto que tendrá la constante en el momento que se utiliza. La segunda opción, orientada al tipado implícito, supone la implementación de un algoritmo de *inferencia de tipos*. Utilizando ideas del *algoritmo Hindley-Milner* [8], se podría adecuar nuestro sistema de tipos para lograr que el mismo sea capaz de inferir los tipos apropiados, sin necesidad de anotar el tipo de nuestra constante.

Bibliografía

[1] Niklaus Wirth. *The Programming Language Pascal (Revised Report)*. Springer Verlag, 1972.

[2] Daan Leijen. *Parsec*. Ver. 3.1.14.0. 2019. URL: https://github.com/haskell/parsec.

[3] Mark Karpov. *Megaparsec*. Ver. 8.0.0. 2019. URL: https://github.com/mrkkrp/megaparsec.

[4] Simon Marlow. *Haskell 2010 Language Report*. Ver. 8.6.5. 2010. URL: https://www.haskell.org/.

[5] GHC Team. *The Glorious Glasgow Haskell Compiler*. Ver. 8.10.1. 2020. URL: https://gitlab.haskell.org/ghc/ghc.

[6] John Charles Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[7] Elias Castegren y Kiko Fernandez Reyes. «Developing a Monadic Type Checker for an Object-Oriented Language: An Experience Report». En: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering* (2019).

[8] Luis Damas y Robin Milner. «Principal Type-Schemes for Functional Programs». En: *9th Symposium on Principles of Programming Languages* (1982).