Semántica Estática

Matias Gobbi

17 de abril de 2020

Fundamentos

Una vez implementada la sintaxis del lenguaje y el parser para el intérprete, es hora de comenzar la etapa del análisis semántico. En este capítulo, describiremos los distintos chequeos estáticos que el intérprete deberá realizar para tener una mayor certeza de que el programa que se quiere ejecutar es correcto. También haremos mención de algunos chequeos dinámicos que puede ser conveniente implementar, debido a que la totalidad de los errores de un programa no puede ser capturada solo con un análisis estático.

1. Introducción

El objetivo de este capítulo es servir como documentación en el desarrollo del intérprete para $\Delta\Delta$Lang. En el mismo, describiremos los distintos aspectos a implementar en el análisis semántico, principalmente los chequeos estáticos y algunos dinámicos. La idea es que el intérprete sea más robusto, y pueda detectar errores en etapas tempranas del desarrollo de un programa. De esta manera, al ejecutar un programa previamente verificado de forma estática, habrá más posibilidades de que su ejecución finalice de forma exitosa.

Mientras que el parser se encarga de filtrar aquellos programas que no estén bien formados sintácticamente, el trabajo de este nuevo análisis es rechazar aquellos fragmentos de código que presenten fallas semánticas, tales como errores de tipos, declaraciones de funciones o procedimientos mal formadas, entre otras. Los chequeos estáticos comprobarán aspectos en una etapa previa a la ejecución, similar a como lo haría un compilador, mientras que los dinámicos lo harán durante la ejecución del mismo programa.

Para dar un formato estructurado al trabajo, el mismo se organizará en base al orden temporal en el que las distintas validaciones se deberán realizar. Se dará una descripción formal para cada uno de los chequeos a implementar, acompañada de una explicación informal para facilitar su comprensión. También se hará mención de distintos aspectos que no fueron definidos todavía, o que aún están sujetos a posibles modificaciones futuras; los mismos están pendientes a ser debatidos en el transcurso del desarrollo del intérprete.

Los fundamentos teóricos utilizados en esta sección están basados en el libro *Theories of Programming Language*, de *John Reynolds*. En particular los capítulos sobre el sistema de tipos (*15*), el subtipado (*16*), y el polimorfismo (*18*), son de fundamental importancia para el desarrollo del trabajo.

2. Sintaxis

La sintaxis del lenguaje ya fue descripta en capítulos anteriores. A pesar de esto, en esta sección se dará un análisis teórico en base a la misma por lo que resulta conveniente abstraerse de detalles propios de su implementación. A continuación, entonces, se describirá la sintaxis abstracta del lenguaje de forma matemática.

2.1. 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.

$$\langle expr \rangle ::= \langle const \rangle \mid \langle fcall \rangle \mid \langle op \rangle \mid \langle var \rangle$$

A su vez, una constante puede tomar alguno de los siguientes valores. Los no terminales *int*, *real*, *bool*, y *char* denotan los conjuntos de valores esperados, mientras que *cname* hace referencia a los identificadores de constantes definidas por el usuario. Los terminales **inf** y **null**, representan al infinito y al puntero nulo respectivamente.

$$\langle const \rangle ::= \langle int \rangle \mid \langle real \rangle \mid \langle bool \rangle \mid \langle char \rangle \mid \langle cname \rangle \mid \textbf{inf} \mid \textbf{null}$$

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

$$\langle fcall \rangle ::= \langle id \rangle \ (\ \langle expr \rangle \ \ldots \ \langle expr \rangle \ )$$

Los operadores del lenguaje están conformados por operados numéricos, booleanos, y de orden e igualdad. Observar que será necesaria la implementación de un chequeo de tipos para asegurar el uso apropiado de los mismos.

$$\langle op \rangle ::= \langle expr \rangle \ \langle bin \rangle \ \langle expr \rangle \mid \langle un \rangle \ \langle expr \rangle$$

$$\langle bin \rangle ::= + \mid - \mid * \mid / \mid \% \mid || \mid \&\& \mid <= \mid >= \mid < \mid > \mid == \mid !=$$

$$\langle un \rangle ::= - \mid !$$

Finalmente, describimos las variables con sus respectivos operadores. Las mismas pueden representar un único valor, un arreglo de varias dimensiones, una tupla con múltiples campos, o un puntero a otra estructura en memoria.

```
⟨ var ⟩ ::= ⟨ id ⟩
          | ⟨ var ⟩ [ ⟨ expr ⟩ ... ⟨ expr ⟩ ]
          | ⟨ var ⟩ . ⟨ fname ⟩
          | ⋆ ⟨ var ⟩
```

2.2. Sentencias

Las sentencias se dividen en las siguientes instrucciones. La composición de la *asignación* y el *while* es bastante simple, por lo que se detallan también a continuación. Notar que el no terminal *sblock* se utiliza para representar la secuencia de instrucciones.

```
⟨ sent ⟩ ::= skip | ⟨ assign ⟩ | ⟨ pcall ⟩ | ⟨ if ⟩ | ⟨ while ⟩ | ⟨ for ⟩

⟨ assign ⟩ ::= ⟨ var ⟩ := ⟨ expr ⟩

⟨ while ⟩ ::= while ⟨ expr ⟩ do ⟨ sblock ⟩

⟨ sblock ⟩ ::= ⟨ sent ⟩ ... ⟨ sent ⟩
```

Para la llamada de procedimientos, se utiliza una sintaxis similar a la empleada para funciones. Además de esto, se encuentran definidos dos métodos especiales exclusivamente para el manejo explícito de memoria del programa.

```
⟨ pcall ⟩ ::= ⟨ id ⟩ ( ⟨ expr ⟩ ... ⟨ expr ⟩ ) | alloc ⟨ var ⟩ | free ⟨ var ⟩
```

La instrucción *if* es bastante compleja en su composición. Además de poder especificar un simple condicional **if**, se pueden agregar otras alternativas **elif**, e incluso una condición final **else**.

```
⟨ if ⟩ ::= if ⟨ expr ⟩ then ⟨ sblock ⟩ ⟨ elif ⟩ ... ⟨ elif ⟩ ⟨ else ⟩

⟨ elif ⟩ ::= elif ⟨ expr ⟩ then ⟨ sblock ⟩

⟨ else ⟩ ::= else ⟨ sblock ⟩ | ϵ
```

Finalmente, otra instrucción que presenta varias opciones es el *for*. Además de especificar rangos ascendentes con **to**, o descendentes con **downto**, también se podrán detallar estructuras iterables con **in**, las cuales deberán tener su mecanismo de iteración implementado.

```
⟨ for ⟩ ::= for ⟨ id ⟩ := ⟨ expr ⟩ to ⟨ expr ⟩ do ⟨ sblock ⟩
          | for ⟨ id ⟩ := ⟨ expr ⟩ downto ⟨ expr ⟩ do ⟨ sblock ⟩
          | for ⟨ id ⟩ in ⟨ expr ⟩ do ⟨ sblock ⟩
```

2.3. Tipos

Los tipos que soporta el lenguaje pueden dividirse en dos categorías, los nativos y los definidos por el usuario. A su vez, un tipo nativo puede ser básico o estructurado. A continuación se detallan los mismos.

```
⟨type⟩ ::= int | real | bool | char
         | ⟨array⟩
         | ⟨pointer⟩
         | ⟨typevar⟩
         | ⟨typedef⟩
```

Del lado de los tipos nativos estructurados, 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 *sname* representa al tamaño variable, cuyo valor deberá ser resuelto durante la ejecución del programa.

```
⟨array⟩ ::= array ⟨size⟩ ... ⟨size⟩ of ⟨type⟩

⟨size⟩ ::= ⟨nat⟩ | ⟨sname⟩

⟨pointer⟩ ::= pointer ⟨type⟩
```

En el caso de las variables de tipo, las mismas poseen su propia clase de identificadores. En cambio, para los tipos definidos, además de su nombre se deben detallar los tipos en los cuales se instanciará. Si el mismo no posee argumentos, el terminal **of** no se deberá especificar.

```
⟨typevar⟩ ::= ⟨typeid⟩

⟨typedef⟩ ::= ⟨tname⟩ of ⟨type⟩ ... ⟨type⟩
```

Para los argumentos de un procedimiento, es necesario especificar el rol que cumplirá cada una de sus entradas. Es decir, si se emplearán para lectura, escritura, o ambas.

```
⟨io⟩ ::= in | out | in/out
```

También existen distintas clases para los tipos del programa. Las mismas representan una especie de interfaz que caracteriza las propiedades que cumplen cada uno de los tipos que las definen.

```
⟨class⟩ ::= Eq | Ord | Iter
```

Finalmente, para la declaración de nuevos tipos por parte del usuario hay tres posibilidades. Se pueden crear tipos enumerados, sinónimos de tipos y tuplas. Para los dos últimos, se pueden especificar parámetros de tipos que permiten crear estructuras más abstractas. Similar a lo dicho anteriormente, si los tipos declarados no poseen argumentos, entonces el terminal **of** se omite.

```
⟨typedecl⟩ ::= enum ⟨tname⟩ = ⟨cname⟩ ... ⟨cname⟩
             | syn ⟨tname⟩ of ⟨typeargs⟩ = ⟨type⟩
             | tuple ⟨tname⟩ of ⟨typeargs⟩ = ⟨field⟩ ... ⟨field⟩

⟨typeargs⟩ ::= ⟨typevar⟩ ... ⟨typevar⟩

⟨field⟩ ::= ⟨fname⟩ : ⟨type⟩
```

2.4. Programas

Para finalizar con la sintaxis del lenguaje, describiremos como se especifica un programa en el mismo. Un programa está compuesto por una serie de definiciones de tipo, seguidas de una serie de declaraciones de rutinas. Cada una de estas rutinas puede ser, o bien una función, o un procedimiento.

```
⟨ prog ⟩ ::= ⟨ typedecl ⟩ ... ⟨ typedel ⟩ ⟨ routdecl ⟩ ... ⟨ routdecl ⟩

⟨ routdecl ⟩ ::= ⟨ fun ⟩ | ⟨ proc ⟩
```

Denominamos bloque al cuerpo de una de estas rutinas. El mismo está conformado primero por una lista de declaraciones de variables, y segundo por una lista de instrucciones. Para declarar una variable solo se tiene que especificar el identificador de la misma, junto con el tipo que posee.

```
⟨ block ⟩ ::= ⟨ vardecl ⟩ ... ⟨ vardecl ⟩ ⟨ sblock ⟩

⟨ vardecl ⟩ ::= 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 del valor que representarán.

```
⟨ fun ⟩ ::= fun ⟨ id ⟩ ( ⟨ funarg ⟩ ... ⟨ funarg ⟩ ) ret ⟨ funret ⟩
            where ⟨ restrictions ⟩
            ⟨ block ⟩

⟨ funarg ⟩ ::= ⟨ id ⟩ : ⟨ type ⟩

⟨ funret ⟩ ::= ⟨ id ⟩ : ⟨ type ⟩
```

Un procedimiento posee una estructura muy similar a la de una función. Las dos diferencias fundamentales con esta, es que el primero no posee retorno ya que no producirá ningún valor como resultado, y que sus argumentos deben especificar que clase de uso se hará con los mismos.

```
⟨ proc ⟩ ::= proc ⟨ id ⟩ ( ⟨ procarg ⟩ ... ⟨ procarg ⟩ )
             where ⟨ restrictions ⟩
             ⟨ block ⟩

⟨ procarg ⟩ ::= ⟨ io ⟩ ⟨ id ⟩ : ⟨ type ⟩
```

Para agregar a lo anterior, debido que los argumentos de una rutina pueden tener tipo variable, es conveniente poder imponer restricciones a los mismos. De esta forma, se pueden crear funciones o procedimientos más abstractos que funcionen para una gran variedad de tipos, y al mismo tiempo, requerir que los mismos sean instancias de ciertas clases.

```
⟨ restrictions ⟩ ::= ⟨ constraint ⟩ ... ⟨ constraint ⟩

⟨ constraint ⟩ ::= ⟨ typevar ⟩ : ⟨ class ⟩ ... ⟨ class ⟩
```

2.5. Metavariables

A lo largo del informe, se utilizarán diversas metavariables (a veces acompañadas de superíndices, o subíndices) para representar distintas clases de construcciones sintácticas. A continuación se listan las mismas, junto con el elemento sintáctico que comúnmente simbolizarán, a menos que se especifique lo contrario en el momento.

θ	$\langle type \rangle$	e	$\langle expr \rangle$
tn	$\langle tname \rangle$	v	$\langle var \rangle$
cn	$\langle cname \rangle$	x, a	$\langle id \rangle$
fn	$\langle fname \rangle$	s	$\langle size \rangle$
tv	$\langle typevar \rangle$	fd	$\langle field \rangle$
td	$\langle typedecl \rangle$	rd	$\langle routdecl \rangle$
vd	$\langle vardecl \rangle$	sb	$\langle sblock \rangle$
io	$\langle io \rangle$	cl	$\langle class \rangle$
rs	$\langle restrictions \rangle$	ct	$\langle const \rangle$
n	$\langle int \rangle$	r	$\langle real \rangle$
b	$\langle bool \rangle$	c	$\langle char \rangle$
$\oplus$	$\langle bin \rangle$	$\ominus$	$\langle un \rangle$

3. Chequeos

Ahora pasamos propiamente a la definición de los distintos chequeos. Avanzaremos progresivamente en el análisis de un programa, a medida que las distintas propiedades sean enunciadas y verificadas.

Según la sintaxis del lenguaje, un programa posee la siguiente forma, donde vale que $n \geq 0$ y $m > 0$.

$$
\begin{gathered}
typedecl_1 \\
\dots \\
typedecl_n \\
routdecl_1 \\
\dots \\
routdecl_m
\end{gathered}
$$

Para asegurar la corrección (parcial) de un programa, se deberán verificar cada una de sus componentes, y en el caso que todas superen su respectiva verificación, se dirá que el programa es correcto (estáticamente). Comúnmente, cuando uno de estos elementos satisfaga las propiedades requeridas por el análisis se dirá que está *bien formado*.

Continuando con la teoría, utilizaremos la siguiente notación para denotar que la construcción sintáctica χ está *bien formada* bajo el contexto π, en base a las reglas de la categoría γ.

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

También denominadas reglas para *juicios de tipado*, estas construcciones a veces producirán resultados luego de finalizado su análisis. En estos casos, se utilizará la siguiente notación, donde ω representa el resultado final obtenido.

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

Otra situación que se suele presentar a la hora del análisis es el uso de múltiples contextos. Debido que comúnmente se deberá almacenar información de distintos índoles para efectuar la verificación, se empleará la siguiente notación en estos escenarios.

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

A lo largo del informe, 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 de análisis anteriores para continuar con la verificación.

3.1. Validaciones en Declaración de Tipos

Las primeras reglas que especificaremos serán sobre las declaraciones de tipo de un programa. Una definición de tipo *typedecl* puede tener alguna de las tres siguientes formas en base a si se desea definir un tipo enumerado, un sinónimo de tipo, o una tupla.

$$typedecl = \begin{cases} \textbf{enum}\ tn = cn_1, cn_2, \ldots, cn_m \\ \textbf{syn}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = \theta \\ \textbf{tuple}\ tn\ \textbf{of}\ tv_1, \ldots, tv_l = fn_1 : \theta_1, \ldots, fn_m : \theta_m \end{cases}$$

Cuando una declaración de tipo esté *bien formada* 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. La estructura de los mismos se detalla a continuación.

Para los tipos enumerados, 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 no pueden ser repetidos, tanto dentro de una definición, como con respecto a las otras.

$$\pi_e = \{(tn, CN) \mid tn \in \langle tname \rangle \land CN \subset \langle cname \rangle\}$$

En el caso de los sinónimos, además de su nombre, se deben guardar sus variables de tipo (es decir, sus argumentos) junto con el tipo al que representa.

En este contexto, se tiene que asegurar que dentro de una definición no se repitan los identificadores para los argumentos.

$$\pi_s = \{(tn, TV, \theta) \mid tn \in \langle tname \rangle \wedge TV \subset \langle typevar \rangle \wedge \theta \in \langle type \rangle\}$$

Finalmente, para las tuplas, tenemos que recordar su nombre, sus argumentos de tipo, y los distintos campos de su definición. En este caso, además de evitar la repetición de variables de tipo, se tiene que asegurar que los nombres de campos sean únicos dentro del cuerpo de la definición.

$$\pi_t = \{(tn, TV, FD) \mid tn \in \langle tname \rangle \wedge TV \subset \langle typevar \rangle \wedge FD \subset \langle field \rangle\}$$

Estos tres contextos se encargarán de almacenar toda la información relacionada con los tipos definidos por el usuario en un programa. 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.

3.1.1. Tipos en Declaración de Tipos

Antes de definir que se entiende por una declaración de tipo *bien formada*, debemos dar las reglas apropiadas para analizar sus tipos y poder garantizar que los mismos sean válidos. El siguiente conjunto de reglas será utilizado en diversas secciones del análisis de un programa; en cada una de estas situaciones se evidenciarán ligeras modificaciones realizadas al mismo con el fin de adecuarlo al chequeo vigente.

Cuando nos encontramos en el entorno de análisis de una declaración de tipo, a la hora de analizar propiamente un tipo, utilizamos la siguiente notación para denotar que el tipo representado por θ es válido en el contexto de los tipos definidos π_e, π_s, y π_t. Permitiendo un poco de abuso de notación, comúnmente haremos referencia al contexto $\pi_{\mathbf{T}}$ para representar a la anterior tripla de contextos. Esta salvedad la tendremos para evitar tener reglas muy complejas, y poder concentrarnos propiamente en las derivaciones.

$$\pi_e, \pi_s, \pi_t \vdash_t \theta$$
$$\pi_{\mathbf{T}} \vdash_t \theta$$

Para deducir lo anterior, necesitamos definir una serie de reglas empleadas en la construcción de pruebas. Con la aplicación sucesiva de las mismas, se pueden construir juicios de tipado para probar la corrección de los tipos de un programa.

Para empezar, todo tipo básico es un tipo correcto. La verificación de los mismos es inmediata, ya que su regla no presenta ninguna premisa.

Regla DT para Tipos: Básicos

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

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 DT para Tipos: Punteros

$$\frac{\pi_{\mathbf{T}} \vdash_t \theta}{\pi_{\mathbf{T}} \vdash_t \textbf{pointer } \theta}$$

Para los arreglos, se necesitarán verificar los tamaños de sus dimensiones, junto con el tipo de los valores que almacenará. Notar que para analizar los primeros no se necesita ninguna información contextual.

Regla DT para Tipos: Arreglos

$$\frac{\vdash_s s_1 \qquad \ldots \qquad \vdash_s s_n \qquad \pi_{\mathbf{T}} \vdash_t \theta}{\pi_{\mathbf{T}} \vdash_t \textbf{array } s_1, \ldots, s_n \textbf{ of } \theta}$$

En primera instancia, una variable de tipo es correcta de forma inmediata. En cambio, para que una declaración de tipo sea adecuada, no deben existir variables de tipo libres en el cuerpo de su definición.

Regla DT para Tipos: Variables de Tipo

$$\frac{}{\pi_{\mathbf{T}} \vdash_t tv}$$

Para los tipos definidos, especificaremos varias reglas. Las primeras serán más simples, se aplicarán a los tipos definidos que no posean argumentos. La idea es comprobar que el tipo utilizado se encuentra definido en alguno de los contextos empleados para la declaración de tipos, sin importar el contenido del cuerpo de su definición.

Regla DT para Tipos: Tipos Enumerados

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

Regla DT para Tipos: Sinónimos sin Argumentos

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

Regla DT para Tipos: Tuplas sin Argumentos

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

En cambio, para los tipos definidos con argumentos, es necesario realizar unas verificaciones adicionales. En particular, se deben validar todos los tipos especificados como argumentos del mismo, y que la cantidad de estos coincida con los de su definición.

Regla DT para Tipos: Sinónimos con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \theta) \in \pi_s \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_1 \qquad \ldots \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_n}{\pi_e, \pi_s, \pi_t \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_n}$$

Regla DT para Tipos: Tuplas con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \{fd_1, \ldots, fd_m\}) \in \pi_t \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_1 \qquad \ldots \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_n}{\pi_e, \pi_s, \pi_t \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_n}$$

Para el caso del tamaño de las dimensiones de un arreglo, solo hay que verificar que el mismo sea un valor natural. Esto se debe a que en esta etapa no se permiten tamaños variables para las mismas.

Regla DT para Tipos: Tamaños Constantes

$$\frac{}{\vdash_s s} \quad \text{cuando } s \in \langle nat \rangle$$

Para ilustrar un poco el uso de las reglas previas, a continuación se muestra un breve ejemplo donde se realiza la prueba de corrección de un tipo del lenguaje. Asumir los siguientes valores para los contextos de tipos definidos.

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

Entonces, con estos contextos, se puede hacer la prueba de corrección del tipo **pointer** *node* **of** A de la siguiente forma. Notar el uso de las reglas para punteros, tuplas, y variables de tipo.

Prueba 1

$$\frac{\dfrac{(node, \{Z\}, \{(elem, Z), (next, \textbf{pointer } node \textbf{ of } Z)\}) \in \pi_t \qquad \dfrac{}{\pi_e, \pi_s, \pi_t \vdash_t A}}{\pi_e, \pi_s, \pi_t \vdash_t node \textbf{ of } A}}{\pi_e, \pi_s, \pi_t \vdash_t \textbf{pointer } node \textbf{ of } A}$$

3.1.2. Variables de Tipo Libres

El anterior conjunto de reglas nos será de gran utilidad para confirmar cuando una declaración de tipo esta *bien formada*. De todas maneras, aún necesitamos especificar otra clase de verificación para garantizar la corrección de estas estructuras. Precisamos de una forma para comprobar que todos los argumentos de tipo en una declaración sean efectivamente empleados en la definición del mismo.

$$FTV : \langle type \rangle \rightarrow \{ \langle typevar \rangle \}$$

Para cumplir nuestro objetivo, se define la función anterior que calculará el conjunto de *variables de tipo* presentes en un tipo. La idea es poder comprobar la

igualdad de los conjuntos de argumentos con los de variables en una declaración de tipo. A continuación su definición dirigida por sintaxis.

$$
\begin{array}{ll}
FTV(\textbf{int}) & = \emptyset \\
FTV(\textbf{real}) & = \emptyset \\
FTV(\textbf{bool}) & = \emptyset \\
FTV(\textbf{char}) & = \emptyset \\
FTV(\textbf{pointer}\ \theta) & = FTV(\theta) \\
FTV(\textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ \theta) & = FTV(\theta) \\
FTV(tv) & = \{tv\} \\
FTV(tn) & = \emptyset \\
FTV(tn\ \textbf{of}\ \theta_1, \ldots, \theta_n) & = FTV(\theta_1) \cup \ldots \cup FTV(\theta_n)
\end{array}
$$

3.1.3. Declaración de Tipos

Una vez definidas las reglas anteriores, ya estamos en condiciones de determinar cuando una definición de tipo está *bien formada*. Comenzando con un contexto vacío, la idea es que se compruebe la validez de todas las declaraciones del programa, una a la vez. Cuando se determina que una declaración esta *bien formada*, se agrega su información al contexto apropiado y se continua con la siguiente. Notar que un tipo definido solo será accesible para las declaraciones posteriores al mismo.

La notación utilizada para estas reglas será la siguiente. Luego del análisis, se producirá un nuevo contexto donde se agrega la información de la definición recientemente verificada. Recordar que durante estas alteraciones de los contextos se deberá respetar las invariantes de consistencia para los mismos.

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

La regla para la definición de tipos enumerados es simple. Siempre y cuando se mantengan las invariantes mencionadas anteriormente, la deducción es inmediata. Un nombre de constante no puede ser repetido en una misma definición, ni tampoco en el resto de las almacenadas en el contexto.

Regla DT: 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, hay que realizar un par de verificaciones. Primero, se tiene que comprobar que el conjunto de variables de tipo empleadas como argumento coincida con el conjunto de variables de tipo utilizadas en la definición del mismo. Segundo, se emplearán las reglas especificadas anteriormente para chequear el tipo en el cuerpo de la definición. Finalmente, se debe respetar la invariante de unicidad para los argumentos.

Regla DT: Sinónimos sin Argumentos

$$\frac{FTV(\theta) = \emptyset \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{syn}\ tn = \theta : \pi_e, \pi'_s, \pi_t}$$

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

Regla DT: Sinónimos con Argumentos

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

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

Las verificaciones para tuplas son similares a las de sinónimos, salvo que se deben adecuar para los múltiples campos de la misma. Hay que revisar la igualdad entre los argumentos de la definición y las variables libres de todos los tipos. Se tiene que analizar cada uno de los mismos para asegurar su corrección. Y se deben respetar las invariantes de unicidad para los nombres de campos, y los argumentos de tipo.

Regla DT: Tuplas sin Argumentos

$$\frac{FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) = \emptyset \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_1 \qquad \ldots \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_m}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

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

Regla DT: Tuplas con Argumentos

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

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

La última regla es la que permite la definición de tipos recursivos. La única posibilidad de declarar un tipo que se define en términos de si mismo es mediante el uso de punteros dentro de tuplas. Por lo tanto, realizando una leve modificación a las reglas anteriores para permitir esta situación, obtenemos lo siguiente. Notar que la definición de tipos recursivos es bastante restrictiva. Solo se permiten utilizar las mismas variables paramétricas que en la definición, e incluso, se las debe especificar en el mismo orden. Las invariantes de consistencia se deben respetar al igual que en las reglas anteriores.

Regla DT: Recursión para Tuplas sin Argumentos

$$\frac{FTV(\theta_1) \cup \ldots \cup FTV(\theta_m) = \emptyset \qquad \theta_i \neq \textbf{pointer}\ tn \implies \pi_e, \pi_s, \pi_t \vdash_t \theta_i}{\pi_e, \pi_s, \pi_t \vdash_{td} \textbf{tuple}\ tn = fn_1 : \theta_1, \ldots, fn_m : \theta_m : \pi_e, \pi_s, \pi'_t}$$

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

Regla DT: Recursión para Tuplas con Argumentos

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

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

Al igual que en la sección anterior, finalizaremos con un ejemplo ilustrativo. Asumir el siguiente valor para el contexto de tuplas definidas; este será el valor final obtenido luego del análisis del ejemplo.

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

Comenzando con los contextos de tipos vacíos, se puede realizar la prueba para un tipo recursivo de la siguiente forma. Notar el uso de las reglas para las variables libres, y la recursión.

Prueba 2

$$\frac{FTV(Z) \cup FTV(\textbf{pointer } node \textbf{ of } Z) = \{Z\} \qquad \overline{\emptyset_e, \emptyset_s, \emptyset_t \vdash_t Z} \qquad \textbf{pointer } node \textbf{ of } Z}{\emptyset_e, \emptyset_s, \emptyset_t \vdash_{td} \textbf{tuple } node \textbf{ of } Z = elem : Z, next : \textbf{pointer } node \textbf{ of } Z : \emptyset_e, \emptyset_s, \pi_t}$$

3.2. Validaciones en Funciones y Procedimientos

A continuación, empezaremos con el análisis para las declaraciones de funciones y procedimientos de un programa. Una rutina *routddecl* puede tener alguna de las dos siguientes formas en base a si define a una función o a un procedimiento.

$$routdecl = \begin{cases} \textbf{fun } f\ (a_1 : \theta_1, \ldots, a_l : \theta_l) \textbf{ ret } a_r : \theta_r \\ \textbf{where } rs \\ \quad block_f \\ \textbf{proc } p\ (io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l) \\ \textbf{where } rs \\ \quad block_p \end{cases}$$

Similar a la declaración de tipos, cada uno de los procedimientos y funciones definidos en el programa es analizado para comprobar su validez. En el caso de estar *bien formado*, su información es almacenada en el contexto adecuado y se prosigue con la declaración siguiente. La estructura de estos contextos se describe a continuación.

Para las funciones, se tiene que almacenar su identificador junto con todos los elementos que la definen. Estos comprenden sus argumentos, su retorno, y sus restricciones para las variables de tipo especificadas dentro de su prototipo.

En este caso, se tiene que asegurar que el nombre utilizado para referirse a una función sea único en el contexto. Al mismo tiempo, los identificadores empleados para sus argumentos no deben repetirse dentro de la definición.

$$\pi_f = \{(f, FA, fr, rs) \mid f \in \langle id \rangle \wedge FA \subset \langle funarg \rangle \wedge \ldots \\ \ldots fr \in \langle funret \rangle \wedge rs \in \langle restrictions \rangle\}$$

El contexto para procedimientos cumple un rol análogo que el empleado para funciones. Debe almacenar los identificadores definidos, juntos con los argumentos y restricciones asociados al mismo. Las invariantes son idénticas al contexto anterior. Un identificador de procedimiento no puede ser utilizado en más de una definición, y tampoco se pueden repetir los nombres para los argumentos.

$$\pi_p = \{(p, PA, rs) \mid p \in \langle id \rangle \wedge PA \subset \langle procarg \rangle \wedge rs \in \langle restrictions \rangle\}$$

Antes de comenzar con la validación de las funciones y procedimientos, es necesario definir algunos contextos auxiliares adicionales. A la hora de analizar el cuerpo de estas estructuras, es fundamental poder llevar un registro de todos los elementos definidos con sus respectivos identificadores.

En el bloque de una definición de función o procedimiento, se pueden declarar variables. Debido a esto, es indispensable llevar un registro con los nombres y los tipos asociados a los mismos. Obviamente, una invariante de este contexto es que no podrá haber más de un par con el mismo identificador de variable.

$$\pi_v \subset \langle id \rangle \times \langle type \rangle$$

En el prototipo de estas rutinas, se pueden introducir límites variables para las dimensiones de los arreglos. Luego, dentro del cuerpo de la función o procedimiento, se podrán declarar arreglos utilizando estos identificadores. Debido a esto, debemos almacenar todos los nombres introducidos.

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

Por último, se permite la definición de rutinas abstractas en el lenguaje. Esto significa que dentro del prototipo de una función o procedimiento se pueden introducir variables de tipo, cuyos tipos concretos serán resueltos durante la ejecución del programa. Con este último contexto se busca recordar todos los tipos polimórficos introducidos.

$$\pi_{tv} \subset \langle typevar \rangle$$

3.2.1. Tipos en Prototipos de Funciones y Procedimientos

Cuando nos encontramos en el entorno de análisis de una declaración de función o procedimiento, la verificación de un tipo difiere a la realizada previamente. Por lo cual, se necesita de otro conjunto de reglas para la validación de los mismos. En el prototipo de estas rutinas se pueden introducir variables de

tipo, y tamaños de arreglos variables; permitiendo de esta forma definiciones polimórficas. Debido a esto, la aplicación de algunas reglas presentan efectos secundarios, ya que se necesita actualizar el contexto adecuado para almacenar la información de las nuevas estructuras definidas.

La nueva notación que utilizaremos será la siguiente. Para analizar un tipo, seguiremos utilizando el contexto de tipos definidos para su verificación. La diferencia en esta nueva etapa, radica en que al finalizar el chequeo de un tipo se obtendrán dos conjuntos distintos como resultado. El primero, consistirá de todos los tamaños variables empleados en el mismo. El segundo, estará conformado por todas las variables de tipo que ocurren en este. La idea es ir acumulando los nuevos elementos encontrados a medida que se avance en el análisis.

$$\pi_{\mathbf{T}} \vdash_t \theta : \pi_{sn}, \pi_{tv}$$

Para el análisis de los tipos básicos, no se agrega ninguna complejidad. La verificación es inmediata, y como no poseen ninguna estructura menor que necesite ser analizada, ambos resultados son vacíos.

Regla F/P para Tipos: Básicos

$$\frac{}{\pi_{\mathbf{T}} \vdash_t \theta : \emptyset_{sn}, \emptyset_{tv}} \quad \text{cuando } \theta \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{char}\}$$

La regla para los punteros es casi idéntica a la especificada en la sección anterior. La diferencia esencial en esta nueva iteración es que el análisis debe propagar el resultado producido por el tipo dentro del puntero.

Regla F/P para Tipos: Punteros

$$\frac{\pi_{\mathbf{T}} \vdash_t \theta : \pi_{sn}, \pi_{tv}}{\pi_{\mathbf{T}} \vdash_t \mathbf{pointer}\ \theta : \pi_{sn}, \pi_{tv}}$$

Para analizar un arreglo, se tienen que reunir todos los resultados producidos al chequear los tamaños del mismo, y luego agregar los obtenidos al verificar el tipo dentro de este. Notar que analizar el tamaño de un arreglo solo puede devolver el conjunto de tamaños variables utilizadas en el mismo.

Regla F/P para Tipos: Arreglos

$$\frac{\vdash_s s_1 : \pi^1_{sn} \qquad \ldots \qquad \vdash_s s_n : \pi^n_{sn} \qquad \pi_{\mathbf{T}} \vdash_t \theta : \pi_{sn}, \pi_{tv}}{\pi_{\mathbf{T}} \vdash_t \mathbf{array}\ s_1, \ldots, s_n\ \mathbf{of}\ \theta : \pi'_{sn}, \pi_{tv}}$$

donde $\pi'_{sn} = \pi^1_{sn} \cup \ldots \cup \pi^n_{sn} \cup \pi_{sn}$.

Al encontrar una variable de tipo durante el análisis, se debe agregar la misma al contexto correspondiente. Al igual que en el conjunto de reglas anterior, la verificación de estos elementos es inmediata. Obviamente, el conjunto de tamaños variables introducidos es vacío.

Regla F/P para Tipos: Variables de Tipo

$$\frac{}{\pi_{\mathbf{T}} \vdash_t tv : \emptyset_{sn}, \{tv\}_{tv}}$$

Al igual que lo hecho anteriormente, dividiremos las reglas para los tipos definidos en dos conjuntos. El primero se utilizará para tipos que no posean argumentos. Notar que las reglas son prácticamente idénticas a las anteriores, salvo que ahora deben devolver como resultado, los conjuntos vacíos.

Regla F/P para Tipos: Tipos Enumerados

$$\frac{(tn, \{cn_1, \ldots, cn_m\}) \in \pi_e}{\pi_e, \pi_s, \pi_t \vdash_t tn : \emptyset_{sn}, \emptyset_{tv}}$$

Regla F/P para Tipos: Sinónimos sin Argumentos

$$\frac{(tn, \emptyset, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t \vdash_t tn : \emptyset_{sn}, \emptyset_{tv}}$$

Regla F/P para Tipos: Tuplas sin Argumentos

$$\frac{(tn, \emptyset, \{fd_1, \ldots, fd_m\}) \in \pi_t}{\pi_e, \pi_s, \pi_t \vdash_t tn : \emptyset_{sn}, \emptyset_{tv}}$$

Nuevamente, las reglas para tipos definidos con argumentos son similares a las detalladas previamente. En esta ocasión, se tienen que acumular todos los contextos producidos al analizar los distintos tipos empleados como argumentos de los mismos

Regla F/P para Tipos: Sinónimos con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \theta) \in \pi_s \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_1 : \pi^1_{sn}, \pi^1_{tv} \qquad \ldots \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_n : \pi^n_{sn}, \pi^n_{tv}}{\pi_e, \pi_s, \pi_t \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_n : \pi'_{sn}, \pi'_{tv}}$$

donde $\pi'_{sn} = \pi^1_{sn} \cup \ldots \cup \pi^n_{sn}$ y $\pi'_{tv} = \pi^1_{tv} \cup \ldots \cup \pi^n_{tv}$.

Regla F/P para Tipos: Tuplas con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \{fd_1, \ldots, fd_m\}) \in \pi_t \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_1 : \pi^1_{sn}, \pi^1_{tv} \qquad \ldots \qquad \pi_e, \pi_s, \pi_t \vdash_t \theta_n : \pi^n_{sn}, \pi^n_{tv}}{\pi_e, \pi_s, \pi_t \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_n : \pi'_{sn}, \pi'_{tv}}$$

donde $\pi'_{sn} = \pi^1_{sn} \cup \ldots \cup \pi^n_{sn}$ y $\pi'_{tv} = \pi^1_{tv} \cup \ldots \cup \pi^n_{tv}$.

Para el caso del tamaño de las dimensiones de un arreglo, ahora tenemos dos situaciones. Igual que antes, si el mismo es un valor natural, el análisis es inmediato. El resultado obtenido con este chequeo es un conjunto vacío.

Regla F/P para Tipos: Tamaños Constantes

$$\frac{}{\vdash_s s : \emptyset_{sn}} \quad \text{cuando } s \in \langle nat \rangle$$

La nueva regla que se agrega es la siguiente. Si el tamaño es variable, entonces debemos devolver su identificador. De esta forma, se podrá utilizar el mismo en el resto del cuerpo de la función o procedimiento.

Regla F/P para Tipos: Tamaños Variables

$$\frac{}{\vdash_s s : \{s\}_{sn}} \quad \text{cuando } s \in \langle sname \rangle$$

Nuevamente, mostraremos un breve ejemplo donde se utilizan las reglas anteriores para realizar un juicio de tipado. Asumir que nos encontramos analizando el tipo de un argumento, en el prototipo de un procedimiento (o función). Notar el uso de las reglas para puntero, arreglos, tamaños variables de dimensiones, y variables de tipo.

Prueba 3

$$\frac{\dfrac{\dfrac{\overline{\vdash_s n : \{n\}_{sn}} \quad \overline{\vdash_s m : \{m\}_{sn}} \quad \overline{\emptyset_{\mathbf{T}} \vdash_t A : \emptyset_{sn}, \{A\}_{tv}}}{\emptyset_{\mathbf{T}} \vdash_t \textbf{array } n, m \textbf{ of } A : \{n, m\}_{sn}, \{A\}_{tv}}}{\emptyset_{\mathbf{T}} \vdash_t \textbf{pointer array } n, m \textbf{ of } A : \{n, m\}_{sn}, \{A\}_{tv}}$$

3.2.2. Restricciones en Funciones y Procedimientos

Al polimorfismo sobre funciones y procedimientos que mencionamos previamente, se lo puede refinar de alguna manera. La idea, es poder especificar algunas características (o mejor dicho, clases) que debe poseer determinada variable de tipo para ser válida en la respectiva implementación.

Estas estructuras sintácticas, al igual que las anteriores, deben ser analizadas para asegurar su corrección. Utilizaremos la siguiente notación para afirmar que la restricción $tv : cl_1, \ldots, cl_l$, es válida bajo el contexto π_{tv}. Este es el mismo contexto que iremos expandiendo a medida que se verifiquen los distintos argumentos de la función o procedimiento correspondiente, incorporando las variables de tipo que ocurran en estos.

$$\pi_{tv} \vdash_c tv : cl_1, \ldots, cl_l$$

Para analizar un listado de restricciones se deben realizar dos tareas. La primera, es asegurar que no haya dos restricciones distintas con el mismo identificador para variables de tipo. Para la segunda, hay que analizar cada una de las restricciones, de forma individual, y confirmar su corrección.

Regla F/P: Restricciones

$$\frac{\forall i, j \in \{1 \dots m\}.i \neq j \implies tv_i \neq tv_j \qquad \pi_{tv} \vdash_c tv_1 : cl_1, \dots, cl_{l_1} \qquad \dots \qquad \pi_{tv} \vdash_c tv_m : cl_1, \dots, cl_{l_m}}{\pi_{tv} \vdash_r tv_1 : cl_1, \dots, cl_{l_1} \ \dots \ tv_m : cl_1, \dots, cl_{l_m}}$$

Para el análisis de un restricción singular, también debemos verificar dos condiciones. Primero, revisar que no se repitan clases dentro de la restricción. Segundo, hay que asegurar que la variable de tipo a la que se le están imponiendo clases, exista efectivamente en el contexto adecuado.

Regla F/P: Restricción

$$\frac{\forall i, j \in \{1 \dots l\}.i \neq j \implies cl_i \neq cl_j \qquad tv \in \pi_{tv}}{\pi_{tv} \vdash_c tv : cl_1, \dots, cl_l}$$

A continuación, daremos un ejemplo para el uso de estas reglas. A diferencia de todas las pruebas mencionadas anteriormente, en este caso, se ilustrará una situación donde no se pueden aplicar las reglas previas para la derivación de un chequeo de corrección.

Prueba 4

$$\frac{A \neq Z \qquad \dfrac{\mathbf{Eq} \neq \mathbf{Ord} \qquad A \in \pi_{tv}}{\{A\}_{tv} \vdash_c A : \mathbf{Eq}, \mathbf{Ord}} \qquad \dfrac{\dots}{\{A\}_{tv} \vdash_c Z : \mathbf{Eq}}}{\{A\}_{tv} \vdash_r A : \mathbf{Eq}, \mathbf{Ord} \mid Z : \mathbf{Eq}}$$

Se puede notar en el ejemplo, que la variable de tipo Z no pertenece al contexto, por lo que no hay ninguna regla que se pueda aplicar. Debido a esto, no se puede continuar con la derivación, ya que la restricción está *mal formada*. Esta situación se puede dar porque en el prototipo de una función (o procedimiento), analizada previamente, no fue introducida la variable de tipo Z.

3.2.3. Prototipos de Funciones y Procedimientos

Ahora ya estamos en condiciones de especificar las reglas de tipado para las funciones y procedimientos de un programa. Dejaremos el análisis del cuerpo de estas estructuras para la siguiente sección, una vez que se haya explicado cómo se obtiene la información contextual para el estudio del mismo.

A la hora de analizar una función o procedimiento, utilizaremos la siguiente notación para denotar que la rutina *routdecl* es válida en base a los contextos de tipos $\pi_{\mathbf{T}}$, al de funciones π_f, y al de procedimientos π_p; extendiendo estos últimos en caso de que la misma satisfaga las propiedades analizadas. Permitiendo un poco de abuso de notación, nos referiremos al contexto $\pi_{\mathbf{R}}$ para representar a los correspondientes contextos de rutinas.

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

La verificación de una declaración de función es compleja. Los contextos de tipos definidos, y los de rutinas, estarán previamente inicializados. Primero tendremos que analizar los argumentos de la misma, donde llevaremos un registro

sobre las variables de tipo, y tamaños variables introducidos en los mismos. También debemos analizar el tipo del retorno. Para esta situación, tendremos que emplear otro conjunto de reglas que será detallado en la sección siguiente. Luego, se deberán analizar las restricciones especificadas en el prototipo. Finalmente, con toda la información contextual obtenida, y extendiendo el contexto correspondiente a funciones para permitir llamadas recursivas en su bloque, se deberá verificar el cuerpo de la rutina.

Regla F/P: Funciones

$$\frac{\forall i \in \{1 \dots l\}.\pi_{\mathbf{T}} \vdash_t \theta_i : \pi^i_{sn}, \pi^i_{tv} \qquad \pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash^*_t \theta_r \qquad \pi_{tv} \vdash_r rs \qquad \pi_{\mathbf{T}}, \overline{\pi}_f, \pi_p, \pi_{sn}, \pi_{tv} \vdash_b block_f}{\pi_{\mathbf{T}}, \pi_f, \pi_p \vdash_{fp} \textbf{fun}\ f\ (a_1 : \theta_1, \dots, a_l : \theta_l)\ \textbf{ret}\ a_r : \theta_r\ \textbf{where}\ rs \quad block_f : \overline{\pi}_f, \pi_p}$$

donde $\pi_{sn} = \pi^1_{sn} \cup \ldots \cup \pi^l_{sn}$ y $\pi_{tv} = \pi^1_{tv} \cup \ldots \cup \pi^l_{tv}$. Además, se tiene que extender al contexto de funciones con el prototipo en cuestión, para permitir las llamadas recursivas $\overline{\pi}_f = (f, \{(a_1, \theta_1), \dots, (a_l, \theta_l)\}, (a_r, \theta_r), rs) \triangleright \pi_f$.

Hay que recordar que se deben respetar las invariantes para la construcción del contexto de funciones. No puede haber más de una definición para el mismo identificador de función, y tampoco pueden existir múltiples argumentos, incluyendo al retorno, con los mismos nombres en una definición particular. A todo esto, se debe sumar una condición adicional. Los identificadores para tamaños variables de arreglos introducidos en el prototipo de la función, deben ser distintos a los utilizados para los nombres de argumentos, incluyendo al retorno.

Para el análisis del retorno de una función, hicimos una salvedad que es conveniente clarificar. Debido que el mismo no puede introducir nuevas variables de tipo, o tamaños para dimensiones de arreglos variables, ya que no se podría inferir su verdadero tipo; no podemos emplear la misma clase de deducción que la utilizada con los argumentos. Por lo tanto, el conjunto de reglas que se utilizarán para analizar el retorno será el empleado en la verificación de tipos para declaración de variables dentro de bloques.

La verificación de una declaración de procedimiento, al igual que para funciones, es compleja. Los contextos de tipos, y rutinas, estarán previamente inicializados. Se tendrán que analizar individualmente todos los argumentos del mismo, almacenando los tamaños variables, y las variables de tipo especificados en estos. Después, se deberán analizar las restricciones especificadas en el prototipo. Y para terminar, con toda la información contextual obtenida, y extendiendo el contexto correspondiente a procedimientos para permitir llamadas recursivas en su cuerpo, se deberá verificar el bloque de la rutina.

Regla F/P: Procedimientos

$$\frac{\forall i \in \{1 \dots l\}.\pi_{\mathbf{T}} \vdash_t \theta_i : \pi^i_{sn}, \pi^i_{tv} \qquad \pi_{tv} \vdash_r rs \qquad \pi_{\mathbf{T}}, \pi_f, \overline{\pi}_p, \pi_{sn}, \pi_{tv} \vdash_b block_p}{\pi_{\mathbf{T}}, \pi_f, \pi_p \vdash_{fp} \textbf{proc}\ p\ (io_1\ a_1 : \theta_1, \dots, io_l\ a_l : \theta_l)\ \textbf{where}\ rs \quad block_p : \pi_f, \overline{\pi}_p}$$

donde $\pi_{sn} = \pi^1_{sn} \cup \ldots \cup \pi^l_{sn}$ y $\pi_{tv} = \pi^1_{tv} \cup \ldots \cup \pi^l_{tv}$. Además, se tiene que extender al contexto de procedimientos con el prototipo en cuestión, para permitir las llamadas recursivas $\overline{\pi}_p = (p, \{(io_1, a_1, \theta_1), \dots, (io_l, a_l, \theta_l)\}, rs) \triangleright \pi_p$.

De forma análoga que para funciones, en esta ocasión se deben respetar las invariantes para la construcción del contexto de procedimientos. Los identificadores de procedimientos pueden ser empleados sólo en una definición, mientras, los nombres de argumentos no pueden ser repetidos en una declaración particular. También se debe sumar la misma condición adicional que antes. Los nombres utilizados para denotar tamaños variables de arreglos que ocurren en el prototipo del procedimiento, deben ser distintos a los empleados para referirse a los argumentos del mismo.

Para finalizar con este segmento del informe, realizaremos un ejemplo de aplicación con las reglas anteriores. Como la demostración de la corrección de un procedimiento (o función) puede ser extensa, asumiremos que el bloque $block_{sort}$ está especificado correctamente, y omitiremos su prueba. Notar el uso de las diversas reglas, como las correspondientes a arreglos y las de restricciones, al mismo tiempo que se van extendiendo los contextos a lo largo del análisis.

Prueba 5

$$\dfrac{\dfrac{\dfrac{}{\vdash_s n : \{n\}_{sn}} \quad \dfrac{}{\emptyset_{\mathbf{T}} \vdash_t A : \emptyset_{sn}, \{A\}_{tv}}}{\emptyset_{\mathbf{T}} \vdash_t \textbf{array } n \textbf{ of } A : \{n\}_{sn}, \{A\}_{tv}} \quad \dfrac{\dfrac{A \in \{A\}_{tv}}{\{A\}_{tv} \vdash_c A : \textbf{Ord}}}{\{A\}_{tv} \vdash_r A : \textbf{Ord}} \quad \dfrac{\ldots}{\emptyset_{\mathbf{T}}, \emptyset_f, \overline{\pi}_p, \{n\}_{sn}, \{A\}_{tv} \vdash_b block_{sort}}}{\emptyset_{\mathbf{T}}, \emptyset_f, \emptyset_p \vdash_{fp} \textbf{proc } sort \ (\textbf{in/out } a : \textbf{array } n \textbf{ of } A) \textbf{ where } A : \textbf{Ord} \quad block_{sort} : \emptyset_f, \overline{\pi}_p}$$

donde $\overline{\pi}_p = \{(sort, \{(\textbf{in/out}, a, \textbf{array } n \textbf{ of } A)\}, \{(A, \{\textbf{Ord}\})\})\}$.

3.3. Validaciones en Bloques

Una vez examinado el prototipo del procedimiento, o de la función, se debe continuar con la verificación y analizar el cuerpo del mismo. Según la sintaxis, un bloque $block_\gamma$ posee la siguiente forma, donde $n \geq 0$ y $m > 0$. El índice γ lo utilizaremos de forma informal, para hacer referencia al identificador de la rutina a la que pertenece.

$$vardecl_1$$
$$\ldots$$
$$vardecl_n$$
$$sent_1$$
$$\ldots$$
$$sent_m$$

Alcanzada esta etapa del análisis, se puede ver que ya se recopiló una gran cantidad de información contextual para el chequeo del bloque. Tenemos a nuestro alcance la información de tipos definidos, las funciones y procedimientos declarados, y las variables, de tipo o de tamaño, introducidas en el prototipo de la actual rutina siendo verificada. Para simplificar la notación, definiremos una serie de elementos auxiliares con el propósito de agilizar el acceso a esta información.

Supongamos que nos encontramos analizando el cuerpo de una función. Puede resultar conveniente poder calcular fácilmente cuales son los identificadores introducidos en el prototipo de la misma. A continuación, se definen un par de funciones que reciben el nombre de la rutina en cuestión, y obtienen el conjunto de identificadores especificados en el encabezado de la misma.

$$Argumentos_{\pi_f} : \langle id \rangle \to \{\langle id \rangle\}$$
$$Argumentos_{\pi_f}(f) = \{a_1, \ldots, a_l\}$$
$$Retorno_{\pi_f} : \langle id \rangle \to \{\langle id \rangle\}$$
$$Retorno_{\pi_f}(f) = \{a_r\}$$

donde $(f, \{(a_1, \theta_1), \ldots, (a_l, \theta_l)\}, (a_r, \theta_r), rs) \in \pi_f$.

En el caso del análisis de un procedimiento, nos encontramos en una situación similar. A diferencia de las funciones, además de querer averiguar cuales son los identificadores introducidos, necesitamos clasificar los mismos en base a la etiqueta de *entrada/salida* con la que fueron especificados. Para esto, se define la siguiente serie de funciones.

$$Inputs_{\pi_p} : \langle id \rangle \to \{\langle id \rangle\}$$
$$Inputs_{\pi_p}(p) = \{a \mid \exists \theta.(\mathbf{in}, a, \theta) \in PA\}$$
$$Outputs_{\pi_p} : \langle id \rangle \to \{\langle id \rangle\}$$
$$Outputs_{\pi_p}(p) = \{a \mid \exists \theta.(\mathbf{out}, a, \theta) \in PA\}$$
$$InOuts_{\pi_p} : \langle id \rangle \to \{\langle id \rangle\}$$
$$InOuts_{\pi_p}(p) = \{a \mid \exists \theta.(\mathbf{in/out}, a, \theta) \in PA\}$$

donde $(p, PA, rs) \in \pi_p$, con $PA = \{(oi_1, a_1, \theta_1), \ldots, (oi_l, a_l, \theta_l)\}$.

Finalmente, también sería importante saber cuales son los identificadores empleados para denotar a las distintas variables declaradas en el programa, junto con los tamaños variables introducidos en el prototipo de la función o procedimiento actualmente en análisis. Notar que la estructura de este conjunto es independiente a la clase de rutina que se esté verificando, solo depende de la información contextual que se posea en el momento.

$$Vars_{\pi_{sn}, \pi_v} = \{v \mid \exists \theta.(v, \theta) \in \pi_v\} \cup \pi_{sn}$$

3.4. Validaciones en Declaración de Variables

Cuando se declara una variable, se debe comprobar que su identificador sea único en el alcance de análisis. En particular, su nombre debe ser distinto a todos los utilizados en los argumentos (y retornos) de la función o procedimiento en cuestión, de los tamaños variables de arreglos introducidos en el prototipo de los mismos, y de otras variables declaradas en el mismo cuerpo. En base a que clase de rutina se esté analizando, el conjunto siguiente estará conformado de maneras

diferentes. Usaremos *NameSpace*, para referirnos al conjunto de identificadores en uso en el alcance actual.

$$NameSpace_f = Argumentos_{\pi_f}(f) \cup Retorno_{\pi_f}(f) \cup Vars_{\pi_{sn},\pi_v}$$
$$NameSpace_p = Inputs_{\pi_p}(p) \cup Ouputs_{\pi_p}(p) \cup InOuts_{\pi_p}(p) \cup Vars_{\pi_{sn},\pi_v}$$

La notación utilizada para esta clase de regla será la siguiente. Tendremos toda la información contextual necesaria, recopilada hasta este momento. Se realizará la verificación adecuada, y una vez terminada, se extenderá el contexto de variables con la información de la declaración actual.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_{tv}, \pi_v \vdash_{vd} vardecl : \pi'_v$$

Ya estamos en condiciones para dar la regla que garantiza la *buena forma* de una declaración de variables. Para determinar que una definición es válida, hay que verificar dos propiedades. Ningún identificador que se intenta declarar, puede estar en uso actualmente, y el tipo de estos elementos debe ser válido. Hay que recordar que la invariante para la construcción del contexto de variables imponía como restricción, la unicidad de los identificadores utilizados para referirse a las mismas.

Regla B: Declaración de Variables

$$\frac{\forall i \in \{1 \ldots l\}.\ x_i \notin NameSpace_\gamma \qquad \pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \theta}{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_{tv}, \pi_v \vdash_{vd} \mathbf{var}\ x_1, \ldots, x_l : \theta : \pi'_v}$$

donde $\pi'_v = (x_1, \theta) \triangleright \ldots \triangleright (x_l, \theta) \triangleright \pi_v$, y γ hace referencia a la rutina global.

3.4.1. Tipos en Declaración de Variables

Nos encontramos en un nuevo entorno de análisis, dentro de una declaración de variables, intentando analizar un tipo. En esta situación, tenemos que permitir solo las variables, de tipo o de tamaño, que fueron introducidas en el prototipo de la función, o procedimiento, global. Debido a esto, una vez más tendremos que dar un conjunto de reglas diferente a las especificados anteriormente. Los cambios tendrán que ver, principalmente, con el agregado de los contextos para tamaños variables, y variables de tipo, a las reglas previas.

La notación empleada se describe a continuación. Tenemos la información de tipos definidos a nuestro alcance, junto con la de variables disponibles. En esta situación, no se producirá ningún resultado o efecto secundario luego de terminada la verificación.

$$\pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \theta$$

Como siempre, todo tipo básico es un tipo correcto. La verificación de los mismos es inmediata, y no se necesitan consultar los contextos acumulados de variables.

Regla B para Tipos: Básicos

$$\frac{}{\pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \theta} \quad \text{cuando } \theta \in \{\mathbf{int}, \mathbf{real}, \mathbf{bool}, \mathbf{char}\}$$

Nuevamente, un puntero es correcto siempre que el tipo al que hace referencia también lo sea. En esta regla, al igual que la anterior, tampoco se necesitan consultar los contextos adicionales.

Regla B para Tipos: Punteros

$$\frac{\pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \theta}{\pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \mathbf{pointer}\ \theta}$$

Para los arreglos, se deben comprobar todos los tamaños de sus dimensiones, junto con el tipo que almacenará. En esta ocasión, para verificar los primeros se hace uso del contexto de variables de tamaño.

Regla B para Tipos: Arreglos

$$\frac{\pi_{sn} \vdash_s s_1 \qquad \ldots \qquad \pi_{sn} \vdash_s s_n \qquad \pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \theta}{\pi_{\mathbf{T}}, \pi_{sn}, \pi_{tv} \vdash_t \mathbf{array}\ s_1, \ldots, s_n\ \mathbf{of}\ \theta}$$

A diferencia de las verificaciones previas, en este análisis, una variable de tipo no será correcta de forma inmediata. Para que el uso de estos elementos sea válido, el mismo debe haber sido introducido previamente en el prototipo del procedimiento, o función, que lo encapsula.

Regla B para Tipos: Variables de Tipo

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

Detallaremos dos conjuntos de reglas diferentes para los tipos definidos. Primero, analizaremos los tipos que no posean argumentos. La verificación es idéntica a la especificada previamente, salvo por el agregado de los nuevos contextos de variables, de tipo y de tamaño.

Regla B para Tipos: Tipos Enumerados

$$\frac{(tn, \{cn_1, \ldots, cn_m\}) \in \pi_e}{\pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t tn}$$

Regla B para Tipos: Sinónimos sin Argumentos

$$\frac{(tn, \emptyset, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t tn}$$

Regla B para Tipos: Tuplas sin Argumentos

$$\frac{(tn, \emptyset, \{fd_1, \ldots, fd_m\}) \in \pi_t}{\pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t tn}$$

En el caso de los tipos definidos que posean argumentos, se realizarán chequeos parecidos a los aplicados en reglas previas. Se debe verificar que el tipo se encuentre declarado, que la cantidad de sus argumentos sea la correcta, y analizar cada uno de estos con los contextos acumulados.

Regla B para Tipos: Sinónimos con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \theta) \in \pi_s \qquad \pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t \theta_1 \qquad \ldots \qquad \pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t \theta_n}{\pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_n}$$

Regla B para Tipos: Tuplas con Argumentos

$$\frac{(tn, \{tv_1, \ldots, tv_n\}, \{fd_1, \ldots, fd_m\}) \in \pi_t \qquad \pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t \theta_1 \qquad \ldots \qquad \pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t \theta_n}{\pi_e, \pi_s, \pi_t, \pi_{sn}, \pi_{tv} \vdash_t tn \textbf{ of } \theta_1, \ldots, \theta_n}$$

Para el tamaño de las dimensiones de un arreglo, hay dos situaciones. Si el mismos es un valor natural, la verificación es inmediata. No se necesita consultar el contexto de tamaños variables.

Regla B para Tipos: Tamaños Constantes

$$\frac{}{\pi_{sn} \vdash_s s} \quad \text{cuando } s \in \langle nat \rangle$$

En cambio, si el tamaño es variable, hay que verificar que el mismo pertenezca al contexto de variables de tamaño. Estos elementos solo pueden ser utilizados en el cuerpo de una rutina, siempre y cuando hayan sido previamente introducidos en el prototipo de la misma.

Regla B para Tipos: Tamaños Variables

$$\frac{s \in \pi_{sn}}{\pi_{sn} \vdash_s s} \quad \text{cuando } s \in \langle sname \rangle$$

A continuación, daremos un ejemplo ilustrativo para ayudar a la comprensión de las últimas reglas descriptas. Asumiendo que nos encontramos dentro del bloque de un procedimiento *sort* (el mismo del ejemplo anterior), podemos verificar una declaración de variables de la siguiente forma. Notar que el conjunto de identificadores en uso $NameSpace_{sort}$, está compuesto por un argumento, y una variable de tamaño.

Prueba 6

$$\frac{aux \notin NameSpace_{sort} = \{a, n\} \qquad \dfrac{A \in \{A\}_{tv}}{\emptyset_{\mathbf{T}}, \{n\}_{sn}, \{A\}_{tv} \vdash_t A}}{\emptyset_{\mathbf{T}}, \emptyset_f, \pi_p, \{n\}_{sn}, \{A\}_{tv}, \emptyset_v \vdash_{vd} \textbf{var } aux : A : \{(aux, A)\}_v}$$

donde $\pi_p = \{(sort, \{(\textbf{in/out}, a, \textbf{array } n \textbf{ of } A)\}, \{(A, \{\textbf{Ord}\})\})\}$.

3.5. Variables Libres

Una vez examinado el listado de declaraciones de variables, se tiene que verificar el conjunto de sentencias del bloque. Antes de comenzar con ese análisis, debemos definir una serie de funciones auxiliares que nos serán de vital importancia para el chequeo de procedimientos y funciones. En particular, lo que necesitamos es una forma de poder distinguir cual es el uso que se hace de las distintas variables que ocurren en un listado de instrucciones.

La primer función que definiremos, se encargará de calcular el conjunto de *variables libres* presentes en un bloque de sentencias. En el cálculo lambda, se denomina *libre* a toda variable que no se encuentra cuantificada. En nuestro caso, debido que solo analizaremos instrucciones (y las expresiones que se encuentren dentro de las mismas), toda ocurrencia de una variable será, en primera instancia, una ocurrencia *libre*.

$$FV_{\langle expr \rangle} : \langle expr \rangle \to \{\ \langle id \rangle\ \} \qquad FV_{\langle sent \rangle} : \langle sent \rangle \to \{\ \langle id \rangle\ \}$$

Para las expresiones, esta función posee un comportamiento trivial. Debido que en el lenguaje no existe ninguna especie de cuantificación, a nivel de expresiones, toda variable en una expresión es una *variable libre*. A continuación, su definición dirigida por sintaxis.

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

En el caso de las sentencias, esta función presenta un comportamiento más interesante con respecto a las expresiones. Esto se debe a la existencia de una instrucción que cumple un rol similar a un cuantificador del cálculo lambda, el iterador *for*. El mismo, define una variable iteradora x que será declarada, inicializada, y modificada por la instrucción en base al rango especificado en la misma sentencia. Además, el alcance de esta variable se limita al bloque de

instrucciones *sb* que acompaña a la sentencia *for*.

$$
\begin{array}{ll}
FV(\textbf{skip}) & = \emptyset \\
FV(v := e) & = FV(v) \cup FV(e) \\
FV(p(e_1, \ldots, e_n)) & = FV(e_1) \cup \ldots \cup FV(e_n) \\
FV(\textbf{alloc}\ v) & = FV(v) \\
FV(\textbf{free}\ v) & = FV(v) \\
FV(\textbf{if}\ e\ \textbf{then}\ sb_1\ \textbf{else}\ sb_2) & = FV(e) \cup FV(sb_1) \cup FV(sb_2) \\
FV(\textbf{while}\ e\ \textbf{do}\ sb) & = FV(e) \cup FV(sb) \\
FV(\textbf{for}\ x := e_1\ \textbf{to}\ e_2\ \textbf{do}\ sb) & = FV(e_1) \cup FV(e_2) \cup (FV(sb) - \{x\}) \\
FV(\textbf{for}\ x := e_1\ \textbf{downto}\ e_2\ \textbf{do}\ sb) & = FV(e_1) \cup FV(e_2) \cup (FV(sb) - \{x\}) \\
FV(\textbf{for}\ x\ \textbf{in}\ e\ \textbf{do}\ sb) & = FV(e) \cup (FV(sb) - \{x\})
\end{array}
$$

La segunda de las funciones que definiremos, se encargará de calcular el conjunto de *variables asignables* presentes en una secuencia de sentencias. Se dice que una variable es *asignable* cuando su valor es modificado durante la ejecución de una instrucción. Observando las definiciones, se puede probar que el conjunto de variables asignables en un bloque siempre estará incluido en el conjunto de variables libres del mismo.

$$AV_{\langle expr \rangle} : \langle expr \rangle \to \{\ \langle id \rangle\ \} \qquad AV_{\langle sent \rangle} : \langle sent \rangle \to \{\ \langle id \rangle\ \}$$

Técnicamente ninguna variable en una expresión es asignable, ya que la evaluación de una nunca debería producir efectos secundarios. Pero debido que una expresión puede ocurrir en una instrucción, la cual tiene el potencial de modificar el estado de las variables que ocurren en la misma, la función tiene la siguiente estructura.

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

La función anterior tiene un comportamiento particular que es importante rescatar. La idea detrás de la definición para variables individuales, arreglos, y tuplas, quedará más clara una vez se describa la parte complementaria de la función. El detalle particular que queremos aclarar, tiene que ver con los punteros, y el problema del *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 distinto que referencia a la misma estructura en memoria. Debido a esta situación, es imposible poder analizar estáticamente cuando el elemento referido por un puntero es modificado. Por lo tanto, se

deberá relegar la tarea de asegurar la correcta modificación de las posiciones de memoria reservadas por el usuario, al análisis dinámico. Entonces, para varias de las verificaciones estáticas que describiremos en las secciones siguientes, se tendrán que diseñar chequeos análogos para realizar durante la ejecución de un programa.

Volviendo a la definición de la función, el caso para sentencias presenta algunas particularidades. Hay varias instrucciones que pueden modificar el estado de una variable. Las mismas comprenden la asignación, las rutinas especiales *alloc* y *free*, y la llamada a procedimientos definidos. Hay que hacer una salvedad adicional para estos últimos. Antes de eso, especificaremos la definición para sentencias de la función.

$$
\begin{aligned}
&AV(\textbf{skip}) && = \emptyset \\
&AV(v := e) && = AV(v) \\
&AV(p(e_1, \ldots, e_n)) && = \{AV(e_i) \mid a_i \in \\
& && \quad \ldots Outputs(p) \cup InOuts(p)\} \\
&AV(\textbf{alloc}\ v) && = AV(v) \\
&AV(\textbf{free}\ v) && = AV(v) \\
&AV(\textbf{if}\ e\ \textbf{then}\ sb_1\ \textbf{else}\ sb_2) && = AV(sb_1) \cup AV(sb_2) \\
&AV(\textbf{while}\ e\ \textbf{do}\ sb) && = AV(sb) \\
&AV(\textbf{for}\ x := e_1\ \textbf{to}\ e_2\ \textbf{do}\ sb) && = AV(sb) - \{x\} \\
&AV(\textbf{for}\ x := e_1\ \textbf{downto}\ e_2\ \textbf{do}\ sb) && = AV(sb) - \{x\} \\
&AV(\textbf{for}\ x\ \textbf{in}\ e\ \textbf{do}\ sb) && = AV(sb) - \{x\}
\end{aligned}
$$

donde $(p, \{(io_1, a_1, \theta_1), \ldots, (io_n, a_n, \theta_n)\}, rs) \in \pi_p$.

Debido que la llamada a un procedimiento toma una serie de expresiones como argumentos, se tiene que realizar un análisis detallado para calcular cuales serán las variables efectivamente modificadas. Esta es la parte donde las etiquetas de *entrada/salida* cumplen un rol fundamental. Todas las expresiones que correspondan, en la definición del procedimiento, a un argumento de salida, serán modificadas. Idealmente, las mismas deberían ser variables. Mientras, el resto de los argumentos que solo sean de entrada, no serán modificados.

Finalmente, la última de las funciones sobre variables que definiremos, se encargará de obtener el conjunto de *variables de lectura* en el cuerpo de una función o procedimiento. Una variable es considerada de *lectura* si es utilizada en la evaluación de una expresión, comúnmente, para el cálculo de alguna computación. Se puede probar que el conjunto de variables de lectura en una secuencia de instrucciones siempre estará incluido en el conjunto de variables libres de la misma, y además, que no necesariamente será disjunto al de variables asignables del bloque.

$$RV_{\langle expr \rangle} : \langle expr \rangle \to \{\ \langle id \rangle\ \} \qquad RV_{\langle sent \rangle} : \langle sent \rangle \to \{\ \langle id \rangle\ \}$$

La definición de la función para expresiones puede ser poco intuitiva. Nuevamente, su análisis se debe hacer teniendo en cuenta que la variable observada

puede ocurrir dentro de una sentencia que potencialmente modificará su valor. La idea, es que el conjunto de variables de lectura debería formar el *complemento* del conjunto de variables asignables, con respecto a las variables libres del programa. Esto no es del todo cierto, ya que se pueden formar expresiones donde las distintas ocurrencias de una misma variable cumplan distintos roles. De todas formas, se puede reconocer fácilmente que existe una analogía entre ambas funciones.

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

Nuevamente, tenemos que hacer mención del problema del *aliasing*. Debido que la verificación del uso apropiado de la memoria es delegado al análisis dinámico, en esta instancia, solo podemos limitarnos a considerar toda variable que ocurre en un acceso de memoria, como una variable de lectura. A diferencia de la definición anterior, donde ninguna variable utilizada en esta clase de operación se podía considerar como asignable.

Por último, describiremos la definición para sentencias de nuestra función. Se puede observar, nuevamente, que hay una analogía entre los comportamientos de las dos últimas funciones. Otra vez, el caso más complejo es el de las llamadas a procedimientos declarados por el usuario. Su definición dirigida por sintaxis se describe a continuación.

$$
\begin{array}{ll}
RV(\textbf{skip}) & = \emptyset \\
RV(v := e) & = RV(v) \cup FV(e) \\
RV(p(e_1, \ldots, e_n)) & = \{FV(e_i) \mid a_i \in Inputs(p) \cup InOuts(p)\} \\
 & \quad \ldots \cup \{RV(e_i) \mid a_i \in Outputs(p)\} \\
RV(\textbf{alloc}\ v) & = RV(v) \\
RV(\textbf{free}\ v) & = RV(v) \\
RV(\textbf{if}\ e\ \textbf{then}\ sb_1\ \textbf{else}\ sb_2) & = FV(e) \cup RV(sb_1) \cup RV(sb_2) \\
RV(\textbf{while}\ e\ \textbf{do}\ sb) & = FV(e) \cup RV(sb) \\
RV(\textbf{for}\ x := e_1\ \textbf{to}\ e_2\ \textbf{do}\ sb) & = FV(e_1) \cup FV(e_2) \cup (RV(sb) - \{x\}) \\
RV(\textbf{for}\ x := e_1\ \textbf{downto}\ e_2\ \textbf{do}\ sb) & = FV(e_1) \cup FV(e_2) \cup (RV(sb) - \{x\}) \\
RV(\textbf{for}\ x\ \textbf{in}\ e\ \textbf{do}\ sb) & = FV(e) \cup (RV(sb) - \{x\})
\end{array}
$$

donde $(p, \{(io_1, a_1, \theta_1), \ldots, (io_n, a_n, \theta_n)\}, rs) \in \pi_p$.

Para el caso de la llamada a procedimientos, entraremos más en detalle. Si una expresión corresponde a un argumento de lectura, entonces todas las variables que ocurran en la misma también serán de lectura. En cambio, cuando una expresión coincide con un argumento de escritura, se tendrá que realizar un

análisis más detallado para determinar cual es el uso que se hace de las distintas variables dentro de esta.

Determinar cuando hay pasaje por valor y cuando por referencia.

3.6. Validaciones en Sentencias

Ya estamos en condiciones para analizar un bloque, y la lista de instrucciones que se encuentran dentro del mismo. Tenemos la información relacionada con los tipos definidos por el usuario, las funciones y procedimientos declarados en el programa, y los elementos introducidos en el prototipo de la rutina que lo encapsula. Además, ya contamos con las herramientas necesarias para verificar las declaraciones de variables dentro del bloque.

Para el caso del análisis de un bloque, utilizaremos la siguiente notación. Los contextos iniciales, serán los que mencionamos previamente. En esta verificación, no se extenderá ningún contexto ya que toda la información dentro del bloque es local a la rutina general. Lo único que se tiene que hacer en esta etapa, es asegurar la corrección de la estructura.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_{tv} \vdash_b block_\gamma$$

Pasando propiamente a la regla, el chequeo de un bloque se describe a continuación. Se tienen que verificar todas las declaraciones de variables, extendiendo el contexto correspondiente luego de analizada cada una de las mismas. Con esta nueva información obtenida y los conjuntos anteriores, exceptuando el contexto de variables de tipos que ya no será necesario en la verificación, se analizará la secuencia de instrucciones de la función o procedimiento general. Por último, la función ϕ se encargará de verificar todas las propiedades sobre las variables empleadas en las sentencias del bloque, haciendo uso de las funciones definidas en la sección previa.

Regla B: Bloques

$$\frac{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_{tv}, \pi_v^0 \vdash_{vd} vd_1 : \pi_v^1 \quad \dots \quad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_{tv}, \pi_v^{n-1} \vdash_{vd} vd_n : \pi_v^n \quad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v^n \vdash_{sb} sb \quad \phi(sb)}{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_{tv} \vdash_b vd_1 \dots vd_n \quad sb}$$

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

Como mencionamos recién, la función ϕ es la encargada de verificar el uso apropiado de las distintas categorías de variables que ocurren en el bloque de sentencias. La misma será detallada en profundidad, luego de especificar las reglas propias para las sentencias.

Finalmente, comenzaremos con el análisis de las sentencias de un procedimiento o función. La notación se describe a continuación. Tendremos todos los contextos declarados hasta el momento a nuestra disposición, exceptuando el correspondiente a variables de tipo. En el análisis, no se producirá ningún efecto secundario, ya que solo debemos verificar la corrección de las sentencias. Para abreviar un poco la especificación de contextos, utilizaremos $\pi_{\mathbf{V}}$ para representar a los conjuntos de variables y tamaños de arreglos. Al mismo tiempo,

comúnmente nos referiremos al contexto $\pi_{\mathbf{P}}$, para referenciar a toda la información acumulada para la realización de este análisis.

$$\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_s sent$$
$$\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{\mathbf{V}} \vdash_s sent$$
$$\pi_{\mathbf{P}} \vdash_s sent$$

Comenzaremos con la secuencia de sentencias. La verificación de la misma es bastante simple. Solo se tienen que analizar todas las instrucciones que la componen, una por una, de forma secuencial.

Regla B: Bloque de Sentencias

$$\frac{\pi_{\mathbf{P}} \vdash_s sent_1 \qquad \ldots \qquad \pi_{\mathbf{P}} \vdash_s sent_m}{\pi_{\mathbf{P}} \vdash_{sb} sent_1 \ldots sent_m}$$

El análisis del *skip* es trivial. Su verificación es inmediata, y no necesita consultar la información de ninguno de los contextos del programa. Su uso es meramente instructivo.

Regla B: Skip

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

Para el análisis de la asignación, se presenta la primer situación interesante para las reglas de sentencias. Se deben verificar sus dos componentes, y asegurar que ambos posean el mismo tipo.

Regla B: Asignación

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

Para el procedimiento especial *alloc*, también debemos realizar un chequeo de tipos. Debido que el mismo se emplea para reservar espacio en memoria para almacenar la estructura referenciada por un puntero, hay que asegurarnos que su argumento sea efectivamente uno.

Regla B: Alloc

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

Para el procedimiento *free* ocurre una situación similar. Se debe verificar que el argumento que recibe, sea efectivamente un tipo de puntero. En este caso, la instrucción se encarga de liberar el espacio de memoria referenciada por el puntero.

Regla B: Free

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

Una instrucción clásica de los lenguaje imperativos es el *while*. Para verificar el mismo, primero se tiene que comprobar que su guarda sea de tipo booleano. Además, hay que analizar la secuencia de instrucciones que forman su cuerpo.

Regla B: While

$$\frac{\pi_{\mathbf{P}} \vdash_e e : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_{sb} sb}{\pi_{\mathbf{P}} \vdash_s \mathbf{while}\ e\ \mathbf{do}\ sb}$$

La instrucción *if* presenta diversas posibilidades para su especificación. Debido a esto, daremos múltiples reglas para su análisis. De todas formas, las derivaciones siguientes siguen una estructura similar. Para cualquier condicional, se debe comprobar que su guarda sea de tipo booleano, y que la secuencia de instrucciones de su cuerpo sea válida.

Regla B: If

$$\frac{\pi_{\mathbf{P}} \vdash_e e : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_{sb} sb}{\pi_{\mathbf{P}} \vdash_s \mathbf{if}\ e\ \mathbf{then}\ sb}$$

Regla B: Else

$$\frac{\pi_{\mathbf{P}} \vdash_e e : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_{sb} sb_1 \qquad \pi_{\mathbf{P}} \vdash_{sb} sb_2}{\pi_{\mathbf{P}} \vdash_s \mathbf{if}\ e\ \mathbf{then}\ sb_1\ \mathbf{else}\ sb_2}$$

Regla B: Elif

$$\frac{\pi_{\mathbf{P}} \vdash_e e_1 : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_e e_2 : \mathbf{bool} \qquad \pi_{\mathbf{P}} \vdash_{sb} sb_1 \qquad \pi_{\mathbf{P}} \vdash_{sb} sb_2 \qquad \pi_{\mathbf{P}} \vdash_{sb} sb_3}{\pi_{\mathbf{P}} \vdash_s \mathbf{if}\ e_1\ \mathbf{then}\ sb_1\ \mathbf{elif}\ e_2\ \mathbf{then}\ sb_2\ \mathbf{else}\ sb_3}$$

La instrucción *for* también posee varias formas de ser especificada, por lo tanto, a continuación detallaremos una serie de reglas para analizar esta sentencia. En primer lugar, describiremos la verificación a realizar para las instrucciones que utilizan límites. Debido que estas sentencias declaran de forma implícita una variable, se debe revisar que la misma se encuentra disponible en el alcance actual. Además, hay que chequear que los tipos de sus límites coincidan, y que estos sean efectivamente enumerables. Finalmente, se debe analizar el bloque de instrucciones del cuerpo de la sentencia.

Regla B: For To

$$\frac{x \notin NameSpace_\gamma \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_e e_1 : \theta \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_e e_2 : \theta \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi'_v \vdash_{sb} sb}{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_s \mathbf{for}\ x := e_1\ \mathbf{to}\ e_2\ \mathbf{do}\ sb}$$

donde $\pi'_v = (x, \theta) \triangleright \pi_v$, y se satisface que el tipo θ es enumerable.

Regla B: For Downto

$$\frac{x \notin NameSpace_\gamma \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_e e_1 : \theta \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_e e_2 : \theta \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi'_v \vdash_{sb} sb}{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_s \textbf{for } x := e_1 \textbf{ downto } e_2 \textbf{ do } sb}$$

donde $\pi'_v = (x, \theta) \triangleright \pi_v$, y se satisface que el tipo θ es enumerable.

Sobre que tipos del lenguaje tendrán la capacidad de ser enumerados, nos limitaremos a un subconjunto de los disponibles en el mismo. Los números enteros, los caracteres, y las enumeraciones definidas por el usuario, serán las únicas construcciones que podrán ser listadas en los límites de la sentencia.

La otra instrucción *for* que debemos analizar es la que permite trabajar con estructuras iterables. Las verificaciones no son muy diferentes a las realizadas previamente. Hay que revisar que el identificador para la variable declarada se encuentra disponible. Asegurar que el tipo de la expresión interna sea efectivamente iterable. Y chequear la corrección de la secuencia de instrucciones del cuerpo de la sentencia.

Regla B: For In

$$\frac{x \notin NameSpace_\gamma \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_e e : \theta \qquad \pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi'_v \vdash_{sb} sb}{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_s \textbf{for } x \textbf{ in } e \textbf{ do } sb}$$

donde $\pi'_v = (x, \theta) \triangleright \pi_v$, y se satisface que el tipo θ es iterable.

Sobre que tipos del lenguaje podrán ser iterados, aún se tiene que tomar una decisión. Actualmente, se considera que ningún tipo nativo del lenguaje posee esta capacidad. Solo los tipos definidos, que implementen algún mecanismo de iteración podrán ser utilizados en esta especie de instrucciones. Lo que no se ha definido aún, es como se debe instanciar un tipo para que el mismo pertenezca a la categoría de estructuras iterables.

El análisis de la llamada a un procedimiento es el más complicado de todos los especificados anteriormente para sentencias. Para el mismo, hay que verificar una serie de propiedades complejas relacionadas con el polimorfismo que admite el procedimiento. Primero, antes de dar la regla, tendremos que definir dos funciones de sustitución que nos auxiliarán en el análisis de esta instrucción.

La función de sustitución para tamaños variables de arreglos se utilizará para hacer coincidir, justamente, los tamaños de las dimensiones de los distintos arreglos. Lo que se busca con esta definición, es poder unificar los tamaños variables de los argumentos especificados en la declaración del procedimiento, contra los tamaños concretos en los tipos de las expresiones pasadas como argumento en la llamada al mismo.

$$\_ \mid \_ : \langle type \rangle \times \Delta_{sn} \to \langle type \rangle$$
$$\Delta_{sn} = \langle sname \rangle \to \langle size \rangle$$

El comportamiento de la función para tipos se describe a continuación. Notar que para el caso de un arreglo, su aplicación se propaga a los tamaños del mismo. Mientras, para los tipos básicos, la sustitución tiene el mismo comportamiento que la función identidad.

$$
\begin{array}{ll}
\textbf{int} \mid \delta_{sn} & = \textbf{int} \\
\textbf{real} \mid \delta_{sn} & = \textbf{real} \\
\textbf{bool} \mid \delta_{sn} & = \textbf{bool} \\
\textbf{char} \mid \delta_{sn} & = \textbf{char} \\
\textbf{pointer}\ \theta \mid \delta_{sn} & = \textbf{pointer}\ (\theta \mid \delta_{sn}) \\
\textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ \theta \mid \delta_{sn} & = \textbf{array}\ (s_1 \mid \delta_{sn}), \ldots, (s_n \mid \delta_{sn})\ \textbf{of}\ (\theta \mid \delta_{sn}) \\
tv \mid \delta_{sn} & = tv \\
tn \mid \delta_{sn} & = tn \\
tn\ \textbf{of}\ \theta_1, \ldots, \theta_n \mid \delta_{sn} & = tn\ \textbf{of}\ (\theta_1 \mid \delta_{sn}), \ldots, (\theta_n \mid \delta_{sn})
\end{array}
$$

La sustitución previa, se tendrá que propagar a los tamaños de los arreglos. Lo que se intenta conseguir con esta definición, es poder reemplazar todos los tamaños variables de los argumentos en la declaración del procedimiento, para hacerlos coincidir con los tamaños actuales en la llamada al mismo.

$$
\begin{array}{lll}
s \mid \delta_{sn} & = s & s \in \langle nat \rangle \\
s \mid \delta_{sn} & = \delta_{sn}(s) & s \in \langle sname \rangle
\end{array}
$$

La función de sustitución para tipos variables, se empleará con un fin similar al anterior. La idea, es poder hacer coincidir todos los tipos variables en los argumentos de la declaración, con los tipos actuales de las distintas expresiones pasadas como argumento en la invocación del procedimiento.

$$
\begin{array}{c}
\_ \mid \_ : \langle type \rangle \times \Delta_{tv} \to \langle type \rangle \\
\Delta_{tv} = \langle typevar \rangle \to \langle type \rangle
\end{array}
$$

El comportamiento de la función se describe a continuación. Se deben reemplazar todas las variables de tipo, según lo que dicta la función de sustitución provista. Similar a la definición anterior, la sustitución se debe propagar para todos los tipos internos, y en el caso de los tipos básicos, no debería realizar

ninguna modificación.

$$
\begin{array}{ll}
\textbf{int} \mid \delta_{tv} & = \textbf{int} \\
\textbf{real} \mid \delta_{tv} & = \textbf{real} \\
\textbf{bool} \mid \delta_{tv} & = \textbf{bool} \\
\textbf{char} \mid \delta_{tv} & = \textbf{char} \\
\textbf{pointer}\ \theta \mid \delta_{tv} & = \textbf{pointer}\ (\theta \mid \delta_{tv}) \\
\textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ \theta \mid \delta_{tv} & = \textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ (\theta \mid \delta_{tv}) \\
tv \mid \delta_{tv} & = \delta_{tv}(tv) \\
tn \mid \delta_{tv} & = tn \\
tn\ \textbf{of}\ \theta_1, \ldots, \theta_n \mid \delta_{tv} & = tn\ \textbf{of}\ (\theta_1 \mid \delta_{tv}), \ldots, (\theta_n \mid \delta_{tv})
\end{array}
$$

Finalmente, ya nos encontramos en condiciones para definir la regla de análisis para la llamada de procedimientos. Se podría decir que tenemos que verificar tres propiedades esenciales. La primera, que el procedimiento se encuentre efectivamente declarado, y que la cantidad de argumentos coincida con los definidos. La segunda, que los tipos actuales en las expresiones se puedan unificar con los tipos reales de los argumentos, mediante alguna función de sustitución. La tercera, que se satisfagan todas las restricciones de clases impuestas para las variables de tipo dentro de la declaración del procedimiento.

Regla B: Procedimientos

$$
\frac{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e e_1 : \theta_1 \qquad \ldots \qquad \pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e e_n : \theta_n}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_s p(e_1, \ldots, e_n)}
$$

Donde el procedimiento se encuentra declarado con la siguiente información, en el contexto correspondiente.

$$(p, \{(io_1, a_1, \theta_1^*), \ldots, (io_n, a_n, \theta_n^*)\}, rs) \in \pi_p$$

Los tipos de los argumentos actuales y reales se pueden igualar, luego de aplicar alguna sustitución válida.

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

Con la misma sustitución previa, se satisfacen todas las restricciones de clases impuestas en la declaración del procedimiento.

$$\forall (tv, \{cl_1, \ldots, cl_m\}) \in rs.\ \delta_{tv}(tv) = \theta \implies \theta \text{ satisface las clases } cl_1, \ldots, cl_m$$

Vamos a entrar un poco más en detalle sobre la última propiedad a verificar, ya que su especificación puede resultar ambigua. Para tener un poco más de control sobre el polimorfismo que admiten las rutinas de un programa, se pueden añadir restricciones a las variables de tipo que ocurren en su prototipo. Debido a esto, al llamar a una función o procedimiento, se tiene que asegurar que estas restricciones se respetan para garantizar la correcta ejecución del programa.

Ya que aún no se ha definido un mecanismo concluyente para generar instancias de determinadas clases para los tipos definidos por el usuario, este aspecto del lenguaje puede resultar un poco oscuro. En base a la naturaleza de un tipo, y al contexto en el que se encuentra posicionado, habrá distintas condiciones para que el mismo pueda satisfacer, o no, una determinada clase del lenguaje. Se dice que un tipo satisface una clase, si cumple alguna de las siguientes reglas.

1. Si es un tipo básico, satisface naturalmente las clases **Eq**, y **Ord**.
2. Si es un tipo estructurado, satisface solo la clase **Eq**.
3. Si es una variable de tipo, satisface una determinada clase, solo si el encabezado de la rutina global se la impone como restricción.
4. Si es un tipo definido enumerado, satisface naturalmente las clases **Eq**, y **Ord**.
5. Si es un sinónimo de tipo definido, satisface las mismas clases que el tipo al que se asocia.
6. Si es una tupla definida, la misma satisface una clase siempre y cuando, se encuentre implementada la instancia de la misma.

Para finalizar con la sección, y para ayudar a entender un poco mejor la última regla, ilustraremos el empleo de la misma con un breve ejemplo. Haciendo uso del mismo procedimiento que introdujimos en ejemplos previos, podemos probar la corrección de una llamada al mismo. Asumiremos que nos encontramos dentro de un procedimiento cualquiera, por ejemplo, dentro de la rutina *main* como en el siguiente fragmento de código.

```
proc main ()
  var a: array 10 of int
  {some initialization of array a}
  sort(a)
end proc
```

Para la prueba de esta sentencia, debemos tener los contextos apropiados inicializados. En este caso, el contexto de procedimientos tendría que tener la información de la rutina *main*, y el procedimiento previamente declarado *sort*, mientras el de variables debe tener los datos asociados al arreglo a. Los demás conjuntos, al no intervenir en la prueba, pueden ser vacíos.

$$\pi_v = \{(a, \textbf{array } 10 \textbf{ of } int))\}$$
$$\pi_p = \{(sort, \{(\textbf{in/out}, a, \textbf{array } n \textbf{ of } A)\}, \{(A, \{\textbf{Ord}\})\}), (main, \emptyset, \emptyset)\}$$

Ahora pasando propiamente a la prueba, tenemos la siguiente aplicación. Debido que aún no detallamos las reglas para la verificación de expresiones, asumiremos la corrección de la variable a y omitiremos su prueba. Las condiciones extras que se deben demostrar se describen al final.

Prueba 7

$$\frac{\dots}{\dfrac{\emptyset_{\mathbf{T}}, \emptyset_f, \pi_p, \emptyset_{sn}, \pi_v \vdash_e a : \mathbf{array}}{\emptyset_{\mathbf{T}}, \emptyset_f, \pi_p, \emptyset_{sn}, \pi_v \vdash_s sort(a)}}$$

Sobre las propiedades adicionales a probar. Se puede ver, inmediatamente, que la cantidad de argumentos en la llamada coinciden con los de la declaración. Sobre la unificación de los tipos, con las siguientes sustituciones se puede observar que se cumple la igualdad necesaria.

$$\delta_{sn}(n) = 10 \quad \delta_{tv}(A) = \mathbf{int}$$
$$\mathbf{array}\ n\ \mathbf{of}\ A \mid \delta_{sn} \mid \delta_{tv} = \mathbf{array}\ 10\ \mathbf{of}\ \mathbf{int}$$

Por último, hay que verificar que se cumplan las restricciones impuestas por el procedimiento invocado. Solo hay una, y la misma se satisface de forma directa. Debido que el tipo de los enteros es básico, los mismos implementan de forma natural la clase de orden. De esta forma, finaliza la verificación de corrección para la llamada del procedimiento.

$$\delta_{tv}(A) = \mathbf{int} \implies \mathbf{int} \text{ satisface la clase } \mathbf{Ord}$$

3.6.1. Función Phi

En esta sección, describiremos las distintas propiedades que la función ϕ deberá verificar dentro de un bloque de sentencias. En base a la función, o procedimiento, que encapsulan a la secuencia de instrucciones a analizar, las propiedades que se deberán chequear sobre la misma serán distintas. Básicamente, esta función busca asegurar el uso adecuado de las diferentes clases de variables que manipula un programa.

A continuación, vamos a extender las funciones previamente definidas para que acepten un listado de instrucciones. De esta forma, podremos realizar su aplicación al bloque entero de sentencias, y al mismo tiempo, logramos flexibilizar un poco la notación para facilitar la lectura de las reglas.

$$FV(sent_1 \dots sent_m) = FV(sent_1) \cup \dots \cup FV(sent_m)$$
$$AV(sent_1 \dots sent_m) = AV(sent_1) \cup \dots \cup AV(sent_m)$$
$$RV(sent_1 \dots sent_m) = RV(sent_1) \cup \dots \cup RV(sent_m)$$

Sea γ el identificador de la rutina que nos encontramos analizando actualmente. La misma puede referir a una función, o a un procedimiento. En base a las distintas variables que se encuentran declaradas en el alcance actual, la función ϕ deberá verificar distintas condiciones. En particular, para el análisis se tendrán que considerar los argumentos de la rutina, los tamaños variables introducidos en su prototipo, y las variables recientemente declaradas.

La primer propiedad que hay que revisar, es la siguiente. Debido que el tamaño de un arreglo es fijo, el mismo no puede ser modificado durante la ejecución de un programa. Por lo tanto, hay que evitar que en las instrucciones, de una función o procedimiento, se intente alterar su valor.

Función Phi: Evitar modificación de tamaños variables.

$$\pi_{sn} \cap AV(sb) = \emptyset$$

Cuando nos encontramos en el cuerpo de una función, debemos realizar unos chequeos adicionales. En particular, no podemos permitir que la variable de retorno no sea modificada en las instrucciones. Debido que una función devuelve un valor, si el mismo no ha sido asignado, no se podría retornar ningún resultado luego de la ejecución de la misma. Hay que tener en cuenta, que estáticamente no se puede verificar si todas las posiciones de memoria alcanzables por una variable fueron debidamente inicializadas en el cuerpo de la función.

Función Phi: Asignar valor al retorno de función.

$$Retorno(\gamma) \subset AV(sb)$$

Además, sumando otra verificación al análisis de funciones, se tiene que chequear que ninguno de los argumentos pasados a la misma sea alterado. Una función no debería producir efectos secundarios en el contexto donde fue invocada, por lo que se debe verificar la siguiente condición.

Función Phi: Evitar modificación de argumentos de función.

$$Argumentos(\gamma) \cap AV(sb) = \emptyset$$

Pasando al análisis de procedimientos, tenemos que realizar algunas verificaciones particulares para los mismos. Por ejemplo, similar a los argumentos de una función, no podemos permitir que las entradas marcadas como *solo lectura* sean modificadas en el cuerpo del mismo.

Función Phi: Etiquetas *in* respetadas en procedimiento.

$$Inputs(\gamma) \cap AV(sb) = \emptyset$$

Para las otras clases de entradas de un procedimiento, tenemos verificaciones análogas a la anterior. Una entrada que se marca como *solo escritura* no puede ser leída en las instrucciones de la rutina. Comúnmente, estas clases de entradas se utilizan para inicializar estructuras externas al procedimiento.

Función Phi: Etiquetas *out* respetadas en procedimiento.

$$Outputs(\gamma) \cap RV(sb) = \emptyset$$

Para finalizar, ya que nos encontramos analizando el uso de variables dentro de funciones y procedimientos, es prudente hacer mención de algunos chequeos dinámicos que puede ser conveniente implementar en el intérprete. Al no poder detectar todos los errores de un programa de forma estática, necesitamos hacer uso de este análisis para poder identificar posibles fallas durante la ejecución de las diversas rutinas.

El primer chequeo se realiza previo a la lectura de una variable. Antes de acceder a la posición de memoria propia de la misma, se debe verificar que un valor haya sido asignado anteriormente en esta. Caso contrario, la operación de acceso podría tener un comportamiento inesperado.

Una segunda validación ha realizar durante la ejecución de un programa, tiene que ver con la verificación que el retorno de una función haya sido completamente asignado. Con esto nos referimos, que todas las posiciones de memoria accesibles por la variable de retorno tengan un valor fijado al finalizar el cálculo de la función.

3.7. Validaciones en Expresiones

En esta sección presentaremos las verificaciones para las expresiones del lenguaje. Las mismas consisten, en esencia, de los distintos chequeos de tipos junto con las reglas de subtipado y unificación apropiadas. La notación utilizada ya fue empleada en la sección anterior, de todas formas, describiremos su significado formalmente. Usando los mismos contextos de análisis que en la verificación de sentencias, se determina que la expresión e es válida bajo los mismos, con el tipo inferido θ, mediante la siguiente notación.

$$\pi_{\mathbf{P}} \vdash_e e : \theta$$

Comenzaremos con la especificación de las reglas de deducción. Para el caso de las constantes del lenguaje, no se presenta ninguna situación compleja. Las reglas son directas, y su resultado es esperable. Recordar que cada una de las siguiente metavariables representa un elemento cualquier de la construcción sintáctica a la que están asociadas.

Regla TC: Valores Constantes

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

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

Cuando se utiliza una constante enumerada, se tiene que consultar el contexto de tipos definidos correspondiente. El tipo resultado de la expresión, corresponderá al nombre empleado en la definición del tipo en el que la constante fue declarada.

Regla TC: Constantes Enumeradas

$$\frac{(tn, \{cn_1, \ldots, cn, \ldots, cn_m\}) \in \pi_e}{\pi_e, \pi_s, \pi_t, \pi_{\mathbf{R}}, \pi_{\mathbf{V}} \vdash_e cn : tn}$$

Para la constante especial **inf**, su deducción es inmediata. En primera instancia, se asume que la misma tiene tipo entero. Luego, con la introducción del subtipado, también podrá ser utilizada como un valor de tipo real. Esta constante se emplea como límite superior, o inferior si se encuentra negada, para los conjuntos numéricos del lenguaje.

Regla TC: Infinito

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

La constante **null** tiene tipo polimórfico. La misma simboliza un puntero que no señala a ninguna posición válida de memoria. Debido a esto, la constante puede pasar como un puntero que señala a cualquier tipo de estructura. Se utiliza principalmente para evitar *dangling pointers*.

Regla TC: Puntero Nulo

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

Para las variables empleadas en una función o procedimiento, hay cuatro reglas diferentes para la deducción de su tipo. En base al contexto en el que fueron introducidas, se tendrá que emplear una u otra de las siguientes inferencias. Comenzando propiamente con las variables declaradas dentro de un bloque, solo hay que consultar el contexto correspondiente.

Regla TC: Variables Declaradas

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

Los tamaños variables introducidos en el prototipo de una función o procedimiento, pueden ser empleados como una variable más en el código. En este caso, se debe verificar que el mismo exista en el contexto adecuado. El tipo inferido para estos elementos, será siempre entero.

Regla TC: Tamaños Variables

$$\frac{s \in \pi_{sn}}{\pi_{\mathbf{T}}, \pi_{\mathbf{R}}, \pi_{sn}, \pi_v \vdash_e s : \mathbf{int}} \quad \text{cuando } s \in \langle sname \rangle$$

Si nos encontramos analizando el bloque de una función f, puede ocurrir que aparezca alguno de los argumentos, o retorno, introducidos en el prototipo de la misma durante la verificación. En este caso, el tipo inferido será el mismo detallado en el encabezado de la rutina.

Regla TC: Variables de Función

$$\frac{(f, \{(a_1, \theta_1), \ldots, (a_l, \theta_l)\}, (a_r, \theta_r), rs) \in \pi_f}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e a_i : \theta_i}$$

durante el análisis del bloque $block_f$.

En el análisis de un procedimiento p, puede ocurrir una situación análoga a la anterior. Al encontrar una variable especificada como entrada del mismo, se infiere el tipo que se asocia a esta en el encabezado del procedimiento.

Regla TC: Variables de Procedimiento

$$\frac{(p, \{(oi_1, a_1, \theta_1), \ldots, (oi_l, a_l, \theta_l)\}, rs) \in \pi_p}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e a_i : \theta_i}$$

durante el análisis del bloque $block_p$.

A continuación, detallaremos las distintas reglas utilizadas para los diversos operadores de variables. Comenzando con los punteros, se debe verificar que la variable que se intenta acceder sea efectivamente uno. El tipo inferido en la deducción será el referenciado por el puntero en la premisa.

Regla TC: Acceso a Puntero

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

Para el acceso a tuplas, la regla es un poco más compleja. Primero, se debe verificar que la variable a la que se intenta acceder sea efectivamente una tupla. Luego, en base a los argumentos declarados en la definición de la tupla, se debe aplicar una sustitución de variables de tipo finita, con respecto a los parámetros de tipo que fueron especificados cuando se introdujo la variable mencionada.

Regla TC: Acceso a Tuplas

$$\frac{\pi_e, \pi_s, \pi_t, \pi_{\mathbf{R}}, \pi_{\mathbf{V}} \vdash_e v : tn \textbf{ of } \theta_1, \ldots, \theta_l \qquad (tn, \{a_1, \ldots, a_l\}, \{(fn_1, \theta_1^*), \ldots, (fn_m, \theta_m^*)\}) \in \pi_t}{\pi_e, \pi_s, \pi_t, \pi_{\mathbf{R}}, \pi_{\mathbf{V}} \vdash_e v.fn_i : (\theta_i^* \mid [a_1 : \theta_1, \ldots, a_l : \theta_l]_{tv})}$$

Finalmente, analizaremos el acceso a arreglos. Al igual que en las reglas anteriores, tenemos que comprobar que la variable con la que operamos sea efectivamente uno. Luego, hay que verificar que todas las expresiones sean de tipo entero, y que la cantidad de las mismas coincida con las dimensiones que posee la estructura. Notar que asegurar que el acceso a un arreglo sea dentro de los límites válidos, solo se puede realizar durante el análisis dinámico.

Regla TC: Acceso a Arreglos

$$\frac{\pi_{\mathbf{P}} \vdash_e v : \textbf{array } s_1, \ldots, s_n \textbf{ of } \theta \qquad \pi_{\mathbf{P}} \vdash_e e_1 : \textbf{int} \qquad \ldots \qquad \pi_{\mathbf{P}} \vdash_e e_n : \textbf{int}}{\pi_{\mathbf{P}} \vdash_e v[e_1, \ldots, e_n] : \theta}$$

Pasando al análisis de los operadores del lenguaje, se puede observar que varios de los mismos se encuentran sobrecargados. Esto quiere decir, que pueden ser utilizados para operar con valores de tipos diferentes, obteniendo resultados distintos en base a los mismos. Comenzando con los operadores numéricos, los mismos pueden ser utilizados para trabajar con enteros y con reales. Luego de introducidas las reglas de subtipado, se podrá notar que las deducciones son aún más flexibles, permitiendo argumentos de tipos distintos.

Regla TC: Operadores Binarios Numéricos

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

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

Regla TC: Operadores Unarios Numéricos

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

Los siguientes operadores que analizaremos serán los booleanos. Las reglas para los mismos son bastante básicas. En esta ocasión, no hay ninguna especie de sobrecarga, ni tampoco se tendrán que usar reglas de subtipado para hacer coincidir los tipos de los operandos.

Regla TC: Operadores Binarios Booleanos

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

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

Regla TC: Operadores Unarios Booleanos

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

Los últimos operadores que estudiaremos serán los de igualdad y orden. Sus argumentos tendrán que tener el mismo tipo, y el resultado será un valor booleano. Estos operadores estarán definidos para una gran variedad de tipos, siempre y cuando, los mismos implementen las clases **Eq**, y **Ord** respectivamente. En secciones anteriores se dio una descripción informal sobre que tipos satisfacen que clases. Teniendo en cuenta esta información previa, las reglas se detallan a continuación.

Regla TC: Operadores de Igualdad

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

donde $\oplus \in \{==, != \}$, y se satisface que el tipo θ es igualable.

Regla TC: Operadores de Orden

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

donde $\oplus \in \{<, >, <=, >=\}$, y se satisface que el tipo θ es ordenable.

Para terminar con las reglas de inferencia de expresiones, solo resta definir la adecuada para las llamadas a funciones. La verificación de estas, es análoga a la de llamadas a procedimientos. Se deben analizar tres propiedades esenciales. Primero, que la función se encuentre declarada, y que la cantidad de argumentos con los que es invocada coincida con los definidos. Segundo, que los tipos actuales en las expresiones se unifiquen con los tipos reales de sus entradas, mediante la aplicación de alguna sustitución. En este caso, la sustitución se tendrá que realizar también al retorno de la función, obteniendo de esta manera, el resultado del análisis. Tercero, que se satisfagan todas las restricciones de clases impuestas para las variables de tipo dentro de la declaración de la función.

Regla TC: Funciones

$$\frac{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e e_1 : \theta_1 \qquad \ldots \qquad \pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e e_n : \theta_n}{\pi_{\mathbf{T}}, \pi_f, \pi_p, \pi_{\mathbf{V}} \vdash_e f(e_1, \ldots, e_n) : \theta_r}$$

Donde la función se encuentra declarado con la siguiente información, en el contexto correspondiente.

$$(f, \{(a_1, \theta_1^*), \ldots, (a_n, \theta_n^*)\}, (a_r, \theta_r^*), rs) \in \pi_f$$

Los tipos de los argumentos actuales y reales se pueden igualar, luego de aplicar alguna sustitución válida.

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

Con la misma sustitución previa, se satisfacen todas las restricciones de clases impuestas en la declaración de la función.

$$\forall (tv, \{cl_1, \ldots, cl_m\}) \in rs.\ \delta_{tv}(tv) = \theta \implies \theta \text{ satisface las clases } cl_1, \ldots, cl_m$$

Una regla fundamental para el tipado de las expresiones tiene que ver con el subtipado. La misma fue mencionada previamente, como una forma de flexibilizar las deducciones de tipo. En el lenguaje, es simplemente la conversión de un valor de tipo **int** a uno de tipo **real**. Esto permite que programas que funcionan para números reales, también lo hagan con enteros. Se especifica a continuación.

Regla TC: Subtipado

$$\frac{\pi_{\mathbf{P}} \vdash_e e : \mathbf{int}}{\pi_{\mathbf{P}} \vdash_e e : \mathbf{real}}$$

Al igual que en la mayoría de las secciones anteriores, finalizaremos con un breve ejemplo. En el mismo, ilustraremos el uso de la regla de subtipado para poder demostrar la corrección de una aplicación de los operadores de igualdad. Como en el ejemplo no tendremos que consultar ninguno de los contextos del programa, asumiremos que $\pi_{\mathbf{P}}$ se encuentra inicializado con un valor razonable.

Prueba 8

$$\dfrac{\dfrac{\dfrac{}{\pi_{\mathbf{P}} \vdash_e 4 : \mathbf{int}}}{\pi_{\mathbf{P}} \vdash_e 4 : \mathbf{real}} \quad \dfrac{}{\pi_{\mathbf{P}} \vdash_e 4{,}0 : \mathbf{real}}}{\pi_{\mathbf{P}} \vdash_e 4 == 4{,}0 : \mathbf{bool}}$$

3.7.1. Unificación

Se puede observar que algunos tipos de nuestro sistema son equivalentes entre sí, a pesar de utilizar construcciones sintácticas diferentes. Un ejemplo podría ser la declaración de un sinónimo de tipo por parte del usuario, junto con el tipo especificado en el cuerpo de su definición. Sintácticamente estos dos elementos serán distintos ya que, el primero se representará solo con su nombre y sus argumentos, mientras que el segundo podrá ser un tipo válido cualquiera del lenguaje. A pesar de esto, es claro que se puede establecer una igualdad semántica entre ambos elementos.

Para el siguiente conjunto de reglas, emplearemos la siguiente notación. Diremos que un tipo θ unifica a otro tipo θ', bajo el contexto de tipos definidos por el usuario $\pi_{\mathbf{T}}$, mediante la siguiente fórmula.

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

Con el siguiente conjunto de reglas se puede observar que la unificación es una relación de equivalencia sobre nuestro sistema de tipos. La misma satisface las propiedades de reflexividad, simetría, y transitividad. A continuación, el listado de deducciones correspondientes.

Regla U: Reflexividad

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

Regla U: Simetría

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

Regla U: Transitividad

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

Quizá la regla más importante que podemos obtener al introducir la unificación en nuestro conjunto de derivaciones, es la siguiente. La misma, especifica que si podemos deducir cierto tipo sobre una expresión cualquiera, y al mismo tiempo, este tipo puede unificar a otro; entonces podemos deducir este último tipo para nuestra expresión inicial.

Regla U: Unificación

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

Obviamente, con solo el conjunto anterior de deducciones no alcanza. Tenemos que dar reglas más concretos para unificar tipos de nuestro sistema. Comenzando con los punteros, la inferencia es bastante simple. Si dos tipos cualquiera unifican, entonces un puntero que referencia a uno de estos, puede unificar a un puntero que señala al otro.

Regla U: 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 interno del arreglo unifica a otro tipo cualquiera, entonces el arreglo inicial puede unificar a otra estructura nueva con el tipo recientemente introducido como tipo interno.

Regla U: Arreglos

$$\frac{\pi_{\mathbf{T}} \vdash_u \theta \sim \theta'}{\pi_{\mathbf{T}} \vdash_u \textbf{array } s_1, \ldots, s_n \textbf{ of } \theta \sim \textbf{array } s_1, \ldots, s_n \textbf{ of } \theta'}$$

Para el uso de tipos definidos, la idea es análoga a la empleada en las reglas previas. Si cada uno de los parámetros puede unificar a otro tipo, entonces el tipo definido podrá unificar a otro con los argumentos modificados.

Regla U: Tipos Definidos

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

Finalmente, la regla para los sinónimos del lenguaje. Esta deducción permite intercambiar libremente el uso del tipo declarado, con el tipo de su definición a lo largo del programa. Esto equivaldría a la creación de un tipo transparente en el lenguaje. Para el caso de los sinónimos sin parámetros, la derivación es directa, ya que solo se tiene que consultar la existencia del mismo en el contexto correspondiente a sinónimos del programa.

Regla U: 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 argumentos, se necesita realizar una transformación adicional. No alcanza con solo verificar que el tipo se encuentre declarado, y que la cantidad de parámetros sea la correcta. Además, se tiene que aplicar una sustitución de variables de tipo finita al cuerpo de su definición, en base a los argumentos pasados al mismo, para obtener el nuevo tipo unificado.

Regla U: Sinónimos con Argumentos

$$\frac{(tn, \{a_1, \ldots, a_n\}, \theta) \in \pi_s}{\pi_e, \pi_s, \pi_t \vdash_u tn \textbf{ of } \theta_1, \ldots, \theta_n \sim (\theta \mid [a_1 : \theta_1, \ldots, a_n : \theta_n]_{tv})}$$

A continuación, daremos un ejemplo ilustrativo para ayudar a la comprensión de las últimas reglas descriptas. Asumiendo que en el lenguaje se han definido dos sinónimos de tipo, *matrix* y *nat*, podemos emplear las deducciones sobre unificación para probar la equivalencia de distintos tipos. En el ejemplo, derivaremos de forma sucesiva un tipo abstracto definido por el usuario, hasta obtener un tipo nativo concreto propio del lenguaje.

Prueba 9

$$\frac{\dfrac{(matrix, \{A\}, \textbf{array } 5,5 \textbf{ of } A) \in \pi_s}{\emptyset_e, \pi_s, \emptyset_t \vdash_u matrix \textbf{ of } nat \sim (\textbf{array } 5,5 \textbf{ of } A \mid [A : nat]_{tv})} \qquad \dfrac{\dfrac{(nat, \emptyset, \textbf{int}) \in \pi_s}{\emptyset_e, \pi_s, \emptyset_t \vdash_u nat \sim \textbf{int}}}{\emptyset_e, \pi_s, \emptyset_t \vdash_u \textbf{array } 5,5 \textbf{ of } nat \sim \textbf{array } 5,5 \textbf{ of int}}}{\emptyset_e, \pi_s, \emptyset_t \vdash_u matrix \textbf{ of } nat \sim \textbf{array } 5,5 \textbf{ of int}}$$

donde $\pi_s = \{(matrix, \{A\}, \textbf{array } 5,5 \textbf{ of } A), (nat, \emptyset, \textbf{int})\}$.

3.8. Validaciones en Programas

Para finalizar con este capítulo, solo resta dar la regla para analizar un programa. En resumen, para asegurar que un programa es válido (estáticamente), se deben realizar dos etapas de análisis. En la primera se tienen que verificar progresivamente todas las declaraciones de tipo en el mismo, acumulando la información obtenida en el contexto de tipos definidos. Luego como segunda etapa, se deben chequear una por una, todas las funciones y procedimientos del programa, almacenando la información conseguida en estos en los contextos adecuados. Inicialmente, se debe comenzar con ambos contextos generales vacíos, y a medida que progrese el análisis, los mismos se irán expandiendo.

Regla P: Programas

$$\frac{\pi^0_{\mathbf{T}} \vdash_{td} td_1 : \pi^1_{\mathbf{T}} \ \ldots \ \pi^{n-1}_{\mathbf{T}} \vdash_{td} td_n : \pi^n_{\mathbf{T}} \qquad \pi^n_{\mathbf{T}}, \pi^0_{\mathbf{R}} \vdash_{fp} rd_1 : \pi^1_{\mathbf{R}} \ \ldots \ \pi^n_{\mathbf{T}}, \pi^{m-1}_{\mathbf{R}} \vdash_{fp} rd_m : \pi^m_{\mathbf{R}}}{\pi^0_{\mathbf{T}}, \pi^0_{\mathbf{R}} \vdash td_1 \ldots td_n \ rd_1 \ldots rd_m}$$

donde los contextos iniciales son vacíos $\pi^0_{\mathbf{T}} = \emptyset = \pi^0_{\mathbf{R}}$.