Semántica

Matias Gobbi

27 de febrero 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 poder determinar que un fragmento de código es un programa válido en el lenguaje. También haremos mención de algunos chequeos dinámicos que puede ser conveniente implementar.

1. Introducción

El objetivo de este 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 compilar (y ejecutar) solo programas válidos del lenguaje.

Mientras que el parser se encarga de filtrar aquellos programas que no estén bien formados sintácticamente, el trabajo de este nuevo análisis es rechazar aquellos fragmentos de código que presenten errores semánticos. Los chequeos estáticos comprobarán aspectos en tiempo de compilación, mientras que los dinámicos lo harán a la hora de la ejecución de un programa.

Para dar un formato estructurado al trabajo, el mismo se dividirá en cuatro secciones correspondientes a cada una de las principales construcciones sintácticas del lenguaje. 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, 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 ser un valor constante, una llamada a función, una operación sobre otras expresiones o una variable con sus respectivos operadores. Se muestra a continuación.

```
⟨expr⟩ → ⟨const⟩ | ⟨fcall⟩ | ⟨op⟩ | ⟨var⟩
```

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.

```
⟨const⟩ → ⟨int⟩ | ⟨real⟩ | ⟨bool⟩ | ⟨char⟩ | ⟨cname⟩ | inf | 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 variables.

```
⟨fcall⟩ → ⟨id⟩ ( [⟨expr⟩] )
```

Los operadores definidos se detallan a continuación. Observar que será necesario la implementación de un chequeo de tipos para asegurar el uso apropiado de los mismos.

```
⟨op⟩ → ⟨expr⟩ ⟨binop⟩ ⟨expr⟩ | ⟨unop⟩ ⟨expr⟩

⟨binop⟩ → + | - | * | / | % | < | >
        | && | || | == | != | <= | >=

⟨unop⟩ → - | !
```

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⟩] ]
      | ⟨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.

```
⟨sent⟩ → skip | ⟨assign⟩ | ⟨pcall⟩ | ⟨if⟩ | ⟨while⟩ | ⟨for⟩

⟨sblock⟩ → [⟨sent⟩]

⟨assign⟩ → ⟨var⟩ := ⟨expr⟩

⟨while⟩ → while ⟨expr⟩ do ⟨sblock⟩
```

Para la llamada a un procedimiento, se detalla de forma similar a las funciones. Además de esto, se encuentran los dos métodos para el manejo de memoria definidos.

```
⟨pcall⟩ → ⟨id⟩ ( [⟨expr⟩] ) | alloc ⟨var⟩ | free ⟨var⟩
```

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

```
⟨if⟩ → if ⟨expr⟩ then ⟨sblock⟩ ⟨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 y descendentes para la iteración, también se pueden detallar estructuras iterables.

```
⟨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 tiene a los arreglos y a los punteros. Para los primeros, hay que especificar como se definen los límites de los mismos. El no terminal *bname* representa a los límites variables.

```
⟨array⟩ → array [⟨range⟩] of ⟨type⟩

⟨range⟩ → ⟨bound⟩ .. ⟨bound⟩

⟨bound⟩ → ⟨int⟩ | ⟨char⟩ | ⟨cname⟩ | ⟨bname⟩

⟨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 especificar los tipos en los cuales se instanciará.

```
⟨typevar⟩ → ⟨typeid⟩

⟨typedef⟩ → ⟨tname⟩ of [⟨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
```

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.

```
⟨typedecl⟩ → enum  ⟨tname⟩ = [⟨cname⟩]
           | syn   ⟨tname⟩ of ⟨typeargs⟩ = ⟨type⟩
           | tuple ⟨tname⟩ of ⟨typeargs⟩ = [⟨field⟩]

⟨typeargs⟩ → [⟨typevar⟩]

⟨field⟩ → ⟨fname⟩: ⟨type⟩
```

2.3.1. Programas

Para finalizar con la sintaxis, describiremos como se especifica un programa en el lenguaje. El mismo está compuesto por una serie de definiciones de tipo, seguidas de una serie de declaraciones de métodos. A su vez, un bloque está conformado por una lista de declaraciones de variables acompañadas por una lista de sentencias.

```
⟨prog⟩ → [⟨typedecl⟩] [⟨methdecl⟩]

⟨block⟩ → [⟨vardecl⟩] ⟨sblock⟩

⟨vardecl⟩ → var [⟨id⟩]: ⟨type⟩
```

Un método puede ser una función o un procedimiento. Ambas poseen un identificador propio, una lista de argumentos, y un bloque de instrucciones que conforman su cuerpo.

```
⟨methdecl⟩ → ⟨fun⟩ | ⟨proc⟩

⟨fun⟩ → fun ⟨id⟩ ( [⟨funarg⟩] ) ret ⟨funret⟩ ⟨block⟩

⟨funarg⟩ → ⟨id⟩: ⟨type⟩

⟨funret⟩ → ⟨id⟩: ⟨type⟩

⟨proc⟩ → proc ⟨id⟩ ( [⟨procarg⟩] ) ⟨block⟩

⟨procarg⟩ → ⟨io⟩ ⟨id⟩: ⟨type⟩
```

3. Chequeos para Programas

Ahora pasamos propiamente a la definición de los distintos chequeos. Un programa P posee la siguiente forma, donde $n \geq 0$ y $m > 0$.

$$typedecl_1$$
$$\dots$$
$$typedecl_n$$
$$methoddecl_1$$
$$\dots$$
$$methoddecl_m$$

Lo primero que se debería realizar es definir los contextos adecuados para almacenar la información correspondientes a los tipos y métodos definidos. A medida que se avance con el análisis de un programa, los mismos se irán llenando con la información pertinente.

$$\pi_{typedecl} = \pi_{enum} \cup \pi_{syn} \cup \pi_{tuple}$$
$$\pi_{method} = \pi_{fun} \cup \pi_{proc}$$

Vamos a decir que un contexto está *bien formado* cuando no posea nombres repetidos entre las estructuras que almacena. A medida que se van construyendo los contextos, esta *invariante* se tiene que satisfacer para garantizar la unicidad de los distintos identificadores empleados.

3.1. Contextos para Declaración de Tipos

El primer análisis que realizaremos será sobre las declaraciones de tipo del programa. Una definición de tipo $typedecl_i$ puede tener alguna de las tres siguientes formas.

$$typedecl_i = \begin{cases} enum\ t_i = c_1, c_2, \dots, c_m \\ syn\ t_i\ of\ a_1, \dots, a_l = \theta \\ tuple\ t_i\ of\ a_1, \dots, a_l = f_1 : \theta_1, \dots, f_m : \theta_m \end{cases} \qquad \forall i \in \{1 \dots n\}$$

Cuando una declaración de tipo esté *bien formada* su información será almacenada en el contexto adecuado. La estructura de los contextos de declaración de tipos se detalla a continuación.

$$\pi_{enum} = \{(t, cs) \mid t \in \langle tname \rangle \wedge cs \subset \langle cname \rangle\}$$
$$\pi_{syn} = \{(t, as, \theta) \mid t \in \langle tname \rangle \wedge as \subset \langle typevar \rangle \wedge \theta \in \langle type \rangle\}$$
$$\pi_{tuple} = \{(t, as, fs) \mid t \in \langle tname \rangle \wedge as \subset \langle typevar \rangle \wedge fs \subset \langle fname \rangle \times \langle type \rangle\}$$

3.2. Chequeos para 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. Primero, necesitamos aumentar el contexto de las declaraciones de tipos de la siguiente manera.

$$\pi_{type} = \pi_{typedecl} \cup \pi_{typevar}$$
$$\pi_{typevar} \subset \langle typevar \rangle$$

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 π_{type}.

$$\pi_{type} \vdash_t \theta$$

Para deducir esto, necesitamos definir una serie de reglas empleadas en la construcción de pruebas. La validez de las mismas se garantiza solamente cuando se dan las condiciones anteriormente mencionadas.

Regla DT para Tipos: Básicos

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

Regla DT para Tipos: Punteros

$$\frac{\pi_{type} \vdash_t \theta}{\pi_{type} \vdash_t pointer\ \theta}$$

Regla DT para Tipos: Arreglos

$$\frac{\pi_{type} \vdash_r r_1 \qquad \ldots \qquad \pi_{type} \vdash_r r_n \qquad \pi_{type} \vdash_t \theta}{\pi_{type} \vdash_t array\ r_1, \ldots, r_n\ of\ \theta}$$

Regla DT para Tipos: Variables de Tipo

$$\frac{\theta \in \pi_{typevar}}{\pi_{type} \vdash_t \theta} \quad \text{cuando } \theta \in \langle typevar \rangle$$

Regla DT para Tipos: Tipos Definidos

$$\frac{\pi_{type} \vdash_t \theta_1 \qquad \ldots \qquad \pi_{type} \vdash_t \theta_n \qquad |A| = n}{\pi_{type} \vdash_t t\ of\ \theta_1, \ldots, \theta_n} \quad \exists A, B.(t, A, B) \in \pi_{typedecl}$$

Para el caso de los rangos de un arreglo hay que definir otra serie de reglas para comprobar la validez de los mismos. En particular, hay que verificar que el tipo de los límites coincida y no permitir rangos vacíos. Notar también que en este entorno de análisis, se prohíben los límites variables.

Regla DT para Tipos: Rangos

$$\frac{\pi_{type} \vdash_b b_{inf} : \theta \qquad \pi_{type} \vdash_b b_{sup} : \theta \qquad b_{inf} \leq b_{sup}}{\pi_{type} \vdash_r b_{inf} \,..\, b_{sup}}$$

Regla DT para Tipos: Límites de Enteros

$$\frac{}{\pi_{type} \vdash_b n : int} \quad \text{cuando } n \in \langle int \rangle$$

Regla DT para Tipos: Límites de Caracteres

$$\frac{}{\pi_{type} \vdash_b c : char} \quad \text{cuando } c \in \langle char \rangle$$

Regla DT para Tipos: Límites de Constantes

$$\frac{\exists A.(t, A) \in \pi_{enum} \wedge e \in A}{\pi_{type} \vdash_b e : t} \quad \text{cuando } e \in \langle cname \rangle$$

3.3. Chequeos para Declaración de Tipos

Una vez definidas las reglas anteriores, ya estamos en condiciones de determinar cuando una definición de tipo está *bien formada*. La idea es que se compruebe la validez de cada una de las declaraciones del programa, una a la vez, agregando las mismas al contexto adecuado en el caso de que estén *bien formadas* y continuando con la siguiente. En caso contrario, se debería detener el análisis del programa y generar un mensaje de error adecuado.

Regla DT: Enumerados

$$\frac{\forall i, j \in \{1 \ldots n\}.i \neq j \implies c_i \neq c_j \qquad \forall i \in \{1 \ldots n\}, (t^*, A) \in \pi_{enum}.c_i \notin A}{\pi_{typedecl} \vdash_{td} enum\ t = c_1, \ldots, c_n}$$

Regla DT: Sinónimos

$$\frac{\pi_{typedecl} \cup \{a_1, \ldots, a_l\} \vdash_t \theta}{\pi_{typedecl} \vdash_{td} syn\ t\ of\ a_1, \ldots, a_l = \theta}$$

donde todos los argumentos de tipo a_i son distintos entre sí.

Regla DT: Tuplas

$$\frac{\forall i \in \{1 \dots m\}.\pi_{typedecl} \cup \{a_1, \dots, a_l\} \vdash_t \theta_i}{\pi_{typedecl} \vdash_{td} tuple\ t\ of\ a_1, \dots, a_l = f_1 : \theta_1, \dots, f_m : \theta_m}$$

donde todos los alias f_i son distintos entre sí, y los argumentos a_i también.

Resta definir una última regla. La misma, es la que permite la definición de tipos recursivos. La única posibilidad de definir 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 la regla anterior para permitir esta situación, obtenemos lo siguiente.

Regla DT: Recursión

$$\frac{\forall i \in \{1 \dots m\}.\pi_{typedecl} \cup \{a_1, \dots, a_l\} \vdash_t \theta_i \vee \theta_i = pointer\ t\ of\ a_1, \dots, a_l}{\pi_{typedecl} \vdash_{td} tuple\ t\ of\ a_1, \dots, a_l = f_1 : \theta_1, \dots, f_m : \theta_m}$$

donde todos los alias f_i son distintos entre sí, y los argumentos a_i también.

3.4. Contextos para Métodos

En la segunda parte del análisis, nos concentraremos en la declaración de métodos. Un método $methoddecl_i$ puede tener alguna de las dos siguientes formas en base a si define a una función o a un procedimiento.

$$methoddecl_i = \begin{cases} fun\ f_i\ (a_1 : \theta_1, \dots, a_l : \theta_l)\ ret\ a_r : \theta_r \\ \quad block_{f_i} \\ proc\ p_i\ (io_1\ a_1 : \theta_1, \dots, io_l\ a_l : \theta_l) \\ \quad block_{p_i} \end{cases} \qquad \forall i \in \{1 \dots m\}$$

Similar a la declaración de tipos, cada uno de los métodos definidos 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. Caso contrario, se detiene la etapa de análisis semántico con un mensaje de error. La estructura de estos contextos se describe a continuación.

$$\pi_{fun} = \{(f, as, r) \mid f \in \langle id \rangle \wedge as \subset \langle id \rangle \times \langle type \rangle \wedge r \in \langle id \rangle \times \langle type \rangle\}$$
$$\pi_{proc} = \{(p, as) \mid p \in \langle id \rangle \wedge as \subset \langle io \rangle \times \langle id \rangle \times \langle type \rangle\}$$

Antes de comenzar con la validación de los métodos, es necesario definir otro contexto más. A la hora de analizar el cuerpo de una función o un procedimiento es fundamental poder llevar un registro de todas las estructuras definidas con sus respectivos identificadores.

$$\pi = \pi_{type} \cup \pi_{method} \cup \pi_{var}$$
$$\pi_{var} = \pi_{loc} \cup \pi_{bound}$$

En este último contexto, se utiliza π_{loc} para almacenar la información relacionada con las variables declaradas en el cuerpo del método. Mientras que se usa π_{bound} para guardar los datos sobre los límites variables de arreglos.

3.5. Chequeos para Métodos

Cuando se comienza con la verificación de un método, el contexto para variables debe estar vacío. El mismo se irá completando a medida que avance el análisis del método. El contexto de tipos deberá estar previamente inicializado, salvo por el posible agregado de variables de tipo. El último contexto, el de métodos, es extendido solamente cuando se verifica por completo la validez de uno de estos.

Regla M: Funciones

$$\frac{\pi_0 \vdash_t \theta_1 : \pi_1 \qquad \ldots \qquad \pi_{l-1} \vdash_t \theta_l : \pi_l \qquad \pi_l \vdash_t \theta_r : \pi_r \qquad \pi_r^* \vdash_{mb} block_f}{\pi_0 \vdash_m fun\ f\ (a_1 : \theta_1, \ldots, a_l : \theta_l)\ ret\ a_r : \theta_r\ block_f}$$

donde todos los identificadores a_i son distintos entre sí, y de los límites variables especificados en los tipos θ_j.

Regla M: Procedimientos

$$\frac{\pi_0 \vdash_t \theta_1 : \pi_1 \qquad \ldots \qquad \pi_{l-1} \vdash_t \theta_l : \pi_l \qquad \pi_l^* \vdash_{mb} block_p}{\pi_0 \vdash_m proc\ p\ (oi_1\ a_1 : \theta_1, \ldots, oi_l\ a_l : \theta_l)\ block_p}$$

donde todos los identificadores a_i son distintos entre sí, y de los límites variables especificados en los tipos θ_j.

Los contextos aumentados π^* en las reglas, representan el agregado del mismo método analizado al contexto actual π. Esto permite la definición de métodos recursivos, los cuales realizan llamadas a sí mismos en su cuerpo.

$$\pi_r^* = \pi_r \cup \{\ (f, \{(a_1, \theta_1), \ldots, (a_l, \theta_l)\}, (a_r, \theta_r))\ \}$$
$$\pi_l^* = \pi_l \cup \{\ (p, \{(oi_1, a_1, \theta_1), \ldots, (oi_l, a_l, \theta_l)\})\ \}$$

3.6. Chequeos para Tipos en Prototipos de Métodos

Cuando nos encontramos en el entorno de análisis de una declaración de método, la verificación de un tipo difiere a la realizada anteriormente, por lo que se necesita de otro conjunto de reglas para su validación. Utilizamos la siguiente notación para denotar que el tipo representado por θ es válido en el contexto π_{i-1}, produciendo como resultado un contexto aumentado π_i.

$$\pi_{i-1} \vdash_t \theta : \pi_i$$

A continuación se detalla el nuevo conjunto de reglas de inferencia para los tipos especificados en los prototipos de métodos.

Regla M para Tipos: Básicos

$$\frac{}{\pi \vdash_t \theta : \pi} \quad \text{cuando } \theta \in \{int, real, bool, char\}$$

Regla M para Tipos: Punteros

$$\frac{\pi_{i-1} \vdash_t \theta : \pi_i}{\pi_{i-1} \vdash_t pointer\ \theta : \pi_i}$$

Regla M para Tipos: Arreglos

$$\frac{\pi_0 \vdash_r r_1 : \pi_1 \qquad \ldots \qquad \pi_{n-1} \vdash_r r_n : \pi_n \qquad \pi_n \vdash_t \theta : \pi_t}{\pi_0 \vdash_t array\ r_1, \ldots, r_n\ of\ \theta : \pi_t}$$

Regla M para Tipos: Variables de Tipo

$$\frac{}{\pi \vdash_t \theta : \pi \cup \{\theta\}} \quad \text{cuando } \theta \in \langle typevar \rangle$$

Regla M para Tipos: Tipos Definidos

$$\frac{\pi_0 \vdash_t \theta_1 : \pi_1 \qquad \ldots \qquad \pi_{n-1} \vdash_t \theta_n : \pi_n \qquad |A| = n}{\pi_0 \vdash_t t\ of\ \theta_1, \ldots, \theta_n : \pi_n}$$

donde se satisface que $\exists A, B.(t, A, B) \in \pi_{typedecl}$.

Para el caso de los rangos de un arreglo también debemos modificar las reglas para soportar la posibilidad de alteración del contexto utilizado. En particular, en el prototipo de un método se permiten especificar límites variables para los arreglos. En primera instancia, el tipo inferido para los mismos sería *nonsense*.

Regla M para Tipos: Límites de Enteros

$$\frac{}{\pi \vdash_b n : int} \quad \text{cuando } n \in \langle int \rangle$$

Regla M para Tipos: Límites de Caracteres

$$\frac{}{\pi \vdash_b c : char} \quad \text{cuando } c \in \langle char \rangle$$

Regla M para Tipos: Límites de Constantes

$$\frac{\exists A.(t, A) \in \pi_{enum} \wedge e \in A}{\pi \vdash_b e : t} \quad \text{cuando } e \in \langle cname \rangle$$

Regla M para Tipos: Límites de Variables

$$\frac{}{\pi \vdash_b v : \mathbf{ns}} \quad \text{cuando } v \in \langle bname \rangle$$

Regla M para Tipos: Rangos Fijos

$$\frac{\pi \vdash_b b_{inf} : \theta \qquad \pi \vdash_b b_{sup} : \theta \qquad b_{inf} \leq b_{sup}}{\pi \vdash_r b_{inf} \mathinner{..} b_{sup} : \pi}$$

donde se cumple que $\theta \in \{int, char\} \cup \langle tname \rangle$.

Las reglas para la inferencia de los rangos con límites variables son mas complejas. Una dificultad que se suma en este nuevo análisis, es que el tipo inferido para un límite variable tiene que adecuarse al almacenado en el contexto de límites. Es decir, si tuviéramos declarado un arreglo con dimensión $[1..n]$ entonces no sería válido tener otro arreglo con dimensión $[Lunes..n]$. Vamos a asumir, como invariante, que el contexto π_{bound} no puede tener más de una entrada con el mismo identificador. Si no se satisface esta condición, entonces estamos frente a un error de tipo.

Regla M para Tipos: Rangos Variables

$$\frac{\pi \vdash_b b_{inf} : \mathbf{ns} \qquad \pi \vdash_b b_{sup} : \mathbf{ns}}{\pi \vdash_r b_{inf} \mathinner{..} b_{sup} : \pi \cup \{(b_{inf}, \mathbf{ns}), (b_{sup}, \mathbf{ns})\}}$$

Regla M para Tipos: Rangos con Límite Inferior Variable

$$\frac{\pi \vdash_b b_{inf} : \mathbf{ns} \qquad \pi \vdash_b b_{sup} : \theta}{\pi \vdash_r b_{inf} \mathinner{..} b_{sup} : \pi^* \cup \{(b_{inf}, \theta)\}} \quad \text{donde } \pi^* = \pi - \{(b_{inf}, \mathbf{ns})\}$$

donde se cumple que $\theta \in \{int, char\} \cup \langle tname \rangle$.

Regla M para Tipos: Rangos con Límite Superior Variable

$$\frac{\pi \vdash_b b_{inf} : \theta \qquad \pi \vdash_b b_{sup} : \mathbf{ns}}{\pi \vdash_r b_{inf} \mathinner{..} b_{sup} : \pi^* \cup \{(b_{sup}, \theta)\}} \quad \text{donde } \pi^* = \pi - \{(b_{sup}, \mathbf{ns})\}$$

donde se cumple que $\theta \in \{int, char\} \cup \langle tname \rangle$.

3.7. Chequeos para Bloques

Una vez examinados los argumentos del método, se debe verificar el cuerpo del mismo. Un bloque $block_\gamma$ posee la siguiente forma, donde $n \geq 0$ y $m > 0$. El índice γ hace referencia al identificador del método en cuestión.

$$\begin{gathered} var\ x_1^1, \ldots, x_{l_1}^1 : \theta^1 \\ \ldots \\ var\ x_1^n, \ldots, x_{l_n}^n : \theta^n \\ sent_1 \\ \ldots \\ sent_m \end{gathered}$$

Alcanzada esta etapa del análisis, se puede ver que ya se recopiló una gran cantidad de información contextual para el chequeo del bloque. 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 función en cuestión, y obtienen el conjunto de identificadores especificados en el encabezado de la misma.

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

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

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 *IO* con la que fueron especificados. Para esto, se definen una serie de funciones.

$$Inputs_{\pi_{proc}} : \langle id \rangle \to \{\langle id \rangle\}$$
$$Inputs_{\pi_{proc}}(p) = \{a \mid \exists \theta.(in, a, \theta) \in A\}$$
$$Outputs_{\pi_{proc}} : \langle id \rangle \to \{\langle id \rangle\}$$
$$Outputs_{\pi_{proc}}(p) = \{a \mid \exists \theta.(out, a, \theta) \in A\}$$
$$IO_{\pi_{proc}} : \langle id \rangle \to \{\langle id \rangle\}$$
$$IO_{\pi_{proc}}(p) = \{a \mid \exists \theta.(in/out, a, \theta) \in A\}$$

donde $(p, A) \in \pi_{proc}$, con $A = \{(oi_1, a_1, \theta_1), \ldots, (oi_l, a_l, \theta_l)\}$.

3.8. Chequeos para 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) del método, en los límites variables de arreglos, y de otras variables declaradas en el mismo cuerpo. En base a que clase de método se esté analizando, el conjunto siguiente estará conformado de maneras diferentes.

$$NameSpace = "identificadores\ en\ uso\ en\ el\ alcance\ actual" \subset \langle id \rangle$$
$$NameSpace_f = Argumentos(f) \cup Retorno(f) \cup \pi_{var}$$
$$NameSpace_p = Inputs(p) \cup Ouputs(p) \cup IO(p) \cup \pi_{var}$$

Una vez definido el conjunto anterior, ya estamos en condiciones para dar la regla que garantiza la *buena forma* de una declaración de variables. Al igual que en todos los analices anteriores, se deberán probar todas las construcciones sintácticas una por una.

Regla B: Declaración de Variables

$$\frac{\forall i \in \{1 \ldots l\}.\ x_i \notin NameSpace \qquad \pi \vdash_t \theta}{\pi \vdash_{vd} var\ x_1, \ldots, x_l : \theta}$$

donde todos los identificadores x_i son distintos entre sí.

3.9. Chequeos para Tipos en Declaración de Variables

Nuevamente, es necesario modificar nuestro conjunto de reglas para la especificación de tipos con el fin de adecuarnos al nuevo contexto de análisis. En esta ocasión, debemos limitar el uso de variables de tipo y de límites de arreglos solo a los introducidos en el prototipo del método analizado. La mayoría de las reglas permanecen sin cambios, salvo que ahora al analizar un tipo no se modificarán los contextos relacionados.

Regla B para Tipos: Básicos

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

Regla B para Tipos: Punteros

$$\frac{\pi \vdash_t \theta}{\pi \vdash_t pointer\ \theta}$$

Regla B para Tipos: Arreglos

$$\frac{\pi \vdash_r r_1 \qquad \ldots \qquad \pi \vdash_r r_n \qquad \pi \vdash_t \theta}{\pi \vdash_t array\ r_1, \ldots, r_n\ of\ \theta}$$

Regla B para Tipos: Variables de Tipo

$$\frac{\theta \in \pi_{typevar}}{\pi \vdash_t \theta} \quad \text{cuando } \theta \in \langle typevar \rangle$$

Regla B para Tipos: Tipos Definidos

$$\frac{\pi \vdash_t \theta_1 \qquad \ldots \qquad \pi \vdash_t \theta_n \qquad |A| = n}{\pi \vdash_t t\ of\ \theta_1, \ldots, \theta_n}$$

donde se satisface que $\exists A, B.(t, A, B) \in \pi_{typedecl}$.

Regla B para Tipos: Límites de Enteros

$$\frac{}{\pi \vdash_b n : int} \quad \text{cuando } n \in \langle int \rangle$$

Regla B para Tipos: Límites de Caracteres

$$\frac{}{\pi \vdash_b c : char} \quad \text{cuando } c \in \langle char \rangle$$

Regla B para Tipos: Límites de Constantes

$$\frac{\exists A.(t, A) \in \pi_{enum} \wedge e \in A}{\pi \vdash_b e : t} \quad \text{cuando } e \in \langle cname \rangle$$

Regla B para Tipos: Límites de Variables

$$\frac{(v, \theta) \in \pi_{bound}}{\pi \vdash_b v : \theta} \quad \text{cuando } v \in \langle bname \rangle$$

Regla B para Tipos: Rangos Fijos

$$\frac{\pi \vdash_b b_{inf} : \theta \qquad \pi \vdash_b b_{sup} : \theta \qquad b_{inf} \leq b_{sup}}{\pi \vdash_r b_{inf} \mathinner{..} b_{sup}}$$

donde se cumple que $\theta \in \{int, char\} \cup \langle tname \rangle$.

Regla B para Tipos: Rangos Variables

$$\frac{\pi \vdash_b b_{inf} : \mathbf{ns} \qquad \pi \vdash_b b_{sup} : \mathbf{ns}}{\pi \vdash_r b_{inf} \mathinner{..} b_{sup}}$$

3.10. Variables Libres

Una vez analizado el listado de declaraciones de variables, se debe verificar el conjunto de sentencias del bloque. Antes de pasar a esta parte, vamos a definir una serie de funciones que nos serán de vital importancia para el chequeo de métodos. En particular, necesitamos una forma de poder distinguir cual es el uso que se hace de las distintas variables utilizadas en una lista de sentencias.

La primera de estas funciones, se encarga de calcular el conjunto de *variables libres* presentes en un bloque de instrucciones. Se denomina *libre* a aquella variable cuyo valor inicial puede determinar el resultado de una computación, o ser modificado durante la misma.

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

Para las expresiones, esta función posee un comportamiento trivial. Esto se debe a que toda variable en una expresión es una *variable libre*. A continuación,

su definición dirigida por sintaxis.

$$
\begin{array}{lll}
FV(c) & = \emptyset & c \in \langle const \rangle \\
FV(f(e_1, \ldots, e_n)) & = FV(e_1) \cup \ldots \cup FV(e_n) & \\
FV(e_1 \oplus e_2) & = FV(e_1) \cup FV(e_2) & \oplus \in \langle binop \rangle \\
FV(\ominus e) & = FV(e) & \ominus \in \langle unop \rangle \\
FV(x) & = \{x\} & \\
FV(v[e_1, \ldots, e_n]) & = FV(v) \cup FV(e_1) \cup \ldots \cup FV(e_n) & \\
FV(v.f) & = FV(v) & \\
FV(\#v) & = FV(v) &
\end{array}
$$

En el caso de las sentencias, esta función presenta un comportamiento más interesante con respecto a las expresiones. Para la instrucción *for*, la variable iteradora i solo puede adoptar los valores comprendidos en el rango especificado en la misma instrucción, y sólo durante la ejecución de la sentencia.

$$
\begin{array}{ll}
FV(skip) & = \emptyset \\
FV(v := e) & = FV(v) \cup FV(e) \\
FV(p(e_1, \ldots, e_n)) & = FV(e_1) \cup \ldots \cup FV(e_n) \\
FV(alloc(v)) & = FV(v) \\
FV(free(v)) & = FV(v) \\
FV(if\ e\ then\ s_1\ else\ s_2) & = FV(e) \cup FV(s_1) \cup FV(s_2) \\
FV(while\ e\ do\ s) & = FV(e) \cup FV(s) \\
FV(for\ i := e_1\ to\ e_2\ do\ s) & = FV(e_1) \cup FV(e_2) \cup (FV(s) - \{i\}) \\
FV(for\ i := e_1\ downto\ e_2\ do\ s) & = FV(e_1) \cup FV(e_2) \cup (FV(s) - \{i\}) \\
FV(for\ i\ in\ e\ do\ s) & = FV(e) \cup (FV(s) - \{i\})
\end{array}
$$

La segunda de estas funciones, se encarga 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. 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. Lo que sucede es que a la hora de analizar una asignación se debe obtener la variable a ser modificada por la sentencia. Por lo tanto, la función tiene el siguiente

comportamiento para expresiones.

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

De vuelta, el caso para sentencias es el más complejo. Hay varias instrucciones que pueden modificar el valor de una variable. Entre las mismas se encuentran la asignación y las distintas clases de procedimientos. Hay que hacer una salvedad extra para estos últimos. Al tomar una serie de expresiones como argumentos, no necesariamente todas las entradas van a ser modificadas, sino solo las que corresponden a una etiqueta *out* o *in/out*.

$$
\begin{array}{ll}
AV(skip) & = \emptyset \\
AV(v := e) & = AV(v) \\
AV(p(e_1, \ldots, e_n)) & = \{AV(e_i) \mid a_i \in Outputs(p) \cup IO(p)\} \\
AV(alloc(v)) & = AV(v) \\
AV(free(v)) & = AV(v) \\
AV(if\ e\ then\ s_1\ else\ s_2) & = AV(s_1) \cup AV(s_2) \\
AV(while\ e\ do\ s) & = AV(s) \\
AV(for\ i := e_1\ to\ e_2\ do\ s) & = AV(s) - \{i\} \\
AV(for\ i := e_1\ downto\ e_2\ do\ s) & = AV(s) - \{i\} \\
AV(for\ i\ in\ e\ do\ s) & = AV(s) - \{i\}
\end{array}
$$

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

Finalmente, la última de las funciones sobre variables se encarga de obtener el conjunto de *variables de lectura* en el cuerpo de un método. Una variable es considerada de *lectura* si su valor es utilizado para el cálculo de alguna computación. El conjunto de variables de lectura en una secuencia de instrucciones siempre estará incluido en el conjunto de variables libres de la misma, pero no necesariamente será disjunto al de variables asignables.

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

La definición de la función para expresiones puede ser poco intuitiva. La idea es que el conjunto de variables de lectura debería formar el *complemento* del conjunto de variables asignables. Esto no es del todo cierto ya que se pueden formar expresiones donde las distintas ocurrencias de una misma variable

cumplan distintos roles.

$$
\begin{array}{lll}
RV(c) & = \emptyset & c \in \langle const \rangle \\
RV(f(e_1, \ldots, e_n)) & = FV(e_1) \cup \ldots \cup FV(e_n) & \\
RV(e_1 \oplus e_2) & = FV(e_1) \cup FV(e_2) & \oplus \in \langle binop \rangle \\
RV(\ominus e) & = FV(e) & \ominus \in \langle unop \rangle \\
RV(x) & = \emptyset & \\
RV(v[e_1, \ldots, e_n]) & = RV(v) \cup FV(e_1) \cup \ldots \cup FV(e_n) & \\
RV(v.f) & = RV(v) & \\
RV(\#v) & = RV(v) &
\end{array}
$$

Por último, la definición para sentencias de nuestra función. Se puede observar, nuevamente, que hay una analogía entre los comportamientos de las dos últimas funciones. Notar que para los procedimientos, todos los argumentos que correspondan a una etiqueta *in* o *in/out* son de acceso por lo que todas sus variables también lo serán. En cambio, para las etiquetas *out*, hay que ser más selectivos sobre cuales variables son efectivamente utilizadas para lectura.

$$
\begin{array}{ll}
RV(skip) & = \emptyset \\
RV(v := e) & = RV(v) \cup FV(e) \\
RV(p(e_1, \ldots, e_n)) & = \{FV(e_i) \mid a_i \in Inputs(p) \cup IO(p)\} \cup \ldots \\
 & \quad \ldots \cup \{RV(e_i) \mid a_i \in Outputs(p)\} \\
RV(alloc(v)) & = RV(v) \\
RV(free(v)) & = RV(v) \\
RV(if\ e\ then\ s_1\ else\ s_2) & = FV(e) \cup RV(s_1) \cup RV(s_2) \\
RV(while\ e\ do\ s) & = FV(e) \cup RV(s) \\
RV(for\ i := e_1\ to\ e_2\ do\ s) & = FV(e_1) \cup FV(e_2) \cup (RV(s) - \{i\}) \\
RV(for\ i := e_1\ downto\ e_2\ do\ s) & = FV(e_1) \cup FV(e_2) \cup (RV(s) - \{i\}) \\
RV(for\ i\ in\ e\ do\ s) & = FV(e) \cup (RV(s) - \{i\})
\end{array}
$$

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

3.11. Chequeos para Sentencias

Ya estamos en condiciones para analizar la secuencia de sentencias que conforman un bloque. Tenemos la información relacionada con los tipos definidos por el usuario, el prototipo del método a analizar y el listado de declaraciones de variables previas a las instrucciones del programa.

Regla B: Bloques

$$
\frac{\pi \vdash_{vd} vardecl_1 \quad \ldots \quad \pi \vdash_{vd} vardecl_n \quad \phi(sblock) \quad \pi \vdash_{sb} sblock}{\pi \vdash_{mb} vardecl_1 \ldots vardecl_n \quad sblock}
$$

La función ϕ es la encargada de verificar el uso apropiado de las distintas categorías de variables empleadas en el bloque de sentencias. En particular queremos ser capaces de evitar el uso de variables no declaradas, la modificación de argumentos, la no asignación de retornos, entre otras situaciones.

Vamos a extender las funciones previamente definidas para que acepten un listado de instrucciones. De esta forma, podremos realizar la verificación al bloque entero de sentencias.

$$FV(s_1 \dots s_m) = FV(s_1) \cup \dots \cup FV(s_m)$$
$$AV(s_1 \dots s_m) = AV(s_1) \cup \dots \cup AV(s_m)$$
$$RV(s_1 \dots s_m) = RV(s_1) \cup \dots \cup RV(s_m)$$

Sea $block_\gamma$ el bloque a analizar, donde γ puede ser el identificador de un procedimiento o de una función. Entonces la función ϕ estará compuesta por distintas partes, donde para satisfacer la misma, se deben cumplir todas las ecuaciones siguientes.

Función Gamma: Uso efectivo de variables.

$$NameSpace_\gamma = FV(sblock)$$

Función Gamma: Uso de variables.

$$\pi_{loc} \subset RV(sblock) \cap AV(sblock)$$

Función Gamma: Evitar modificación de límites.

$$\pi_{bound} \cap AV(sblock) = \emptyset$$

Cuando γ hace referencia al identificador de una función, también es necesario verificar el siguiente conjunto de ecuaciones.

Función Gamma: Evitar modificación de argumentos.

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

Función Gamma: Asignar valor al retorno.

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

En cambio, cuando γ representa al identificador de un procedimiento, es necesario revisar otro conjunto de ecuaciones.

Función Gamma: Etiquetas *in* respetadas.

$$Inputs(\gamma) \subset RV(sblock) \quad \wedge \quad Inputs(\gamma) \cap AV(sblock) = \emptyset$$

Función Gamma: Etiquetas *out* respetadas.

$$Outputs(\gamma) \subset AV(sblock) \quad \wedge \quad Outputs(\gamma) \cap RV(sblock) = \emptyset$$

Función Gamma: Etiquetas *in/out* respetadas.

$$IO(\gamma) \subset RV(sblock) \cap AV(sblock)$$

Ahora definiremos propiamente las reglas de inferencia para las sentencias. Las mismas no presentan ninguna complejidad adicional en comparación a lo que se estuvo analizando hasta el momento. En esta sección se presenta una de las decisiones pendientes a tomar para el futuro desarrollo del intérprete.

Regla B: Bloque de Sentencias

$$\frac{\pi \vdash_s sent_1 \qquad \ldots \qquad \pi \vdash_s sent_m}{\pi \vdash_{sb} sent_1 \ldots sent_m}$$

Regla B: Skip

$$\frac{}{\pi \vdash_s skip}$$

Regla B: Asignación

$$\frac{\pi \vdash_e v : \theta \qquad \pi \vdash_e e : \theta}{\pi \vdash_s v := e}$$

Cuando se invoca un procedimiento, además de comprobar la existencia del mismo, se necesita realizar una verificación adicional. Debido a que un procedimiento puede modificar sus entradas, es necesario chequear que las expresiones correspondientes a las mismas sean variables para que efectivamente suceda el cambio de estado.

Regla B: Procedimientos

$$\frac{\pi \vdash_e e_1 : \theta_1 \qquad \ldots \qquad \pi \vdash_e e_n : \theta_n}{\pi \vdash_s p(e_1, \ldots, e_n)}$$

donde se satisfacen las siguientes propiedades.

$$(p, \{(io_1, a_1, \theta_1), \ldots, (io_n, a_n, \theta_n)\}) \in \pi_{proc}$$
$$\forall i \in \{1 \ldots n\}. a_i \in Outputs(p) \cup IO(p) \implies e_i \in \langle var \rangle$$

Regla B: Alloc

$$\frac{\pi \vdash_e v : pointer\ \theta}{\pi \vdash_s alloc(v)} \quad \text{con } \theta \in \langle TypeSystem \rangle$$

Regla B: Free

$$\frac{\pi \vdash_e v : pointer\ \theta}{\pi \vdash_s free(v)} \quad \text{con } \theta \in \langle TypeSystem \rangle$$

Regla B: While

$$\frac{\pi \vdash_e e : bool \qquad \pi \vdash_{sb} s}{\pi \vdash_s while\ e\ do\ s}$$

Regla B: If

$$\frac{\pi \vdash_e e : bool \qquad \pi \vdash_{sb} s_1 \qquad \pi \vdash_{sb} s_2}{\pi \vdash_s if\ e\ then\ s_1\ else\ s_2}$$

Finalmente, especificamos las reglas para las sentencias *for*. Todavía queda pendiente definir como se implementarán las *typeclasses* en el lenguaje. En las sucesivas iteraciones de desarrollo del intérprete se tomará una decisión respecto al asunto. Debido a esto, las siguientes pruebas pueden resultar ambiguas.

Regla B: For To

$$\frac{i \notin NameSpace \qquad i \notin AV(s) \qquad \pi \vdash_e e_1 : \theta \qquad \pi \vdash_e e_2 : \theta \qquad \pi \vdash_{sb} s}{\pi \vdash_s for\ i := e_1\ to\ e_2\ do\ s}$$

donde se satisface que el tipo θ es enumerable.

Regla B: For Downto

$$\frac{i \notin NameSpace \qquad i \notin AV(s) \qquad \pi \vdash_e e_1 : \theta \qquad \pi \vdash_e e_2 : \theta \qquad \pi \vdash_{sb} s}{\pi \vdash_s for\ i := e_1\ downto\ e_2\ do\ s}$$

donde se satisface que el tipo θ es enumerable.

Regla B: For In

$$\frac{i \notin NameSpace \qquad i \notin AV(s) \qquad \pi \vdash_e e : \theta \qquad \pi \vdash_{sb} s}{\pi \vdash_s for\ i\ in\ e\ do\ s}$$

donde se satisface que el tipo θ es iterable.

3.12. Chequeos para Expresiones

Para finalizar, es hora de dar las verificaciones para las expresiones del lenguaje. Las mismas consisten, en esencia, de los distintos chequeos de tipos. Lo primero que debemos hacer es extender nuestra sintaxis para poder especificar una nueva construcción, el *TypeSystem*. La misma no formará parte del lenguaje, sino más bien, será una herramienta que utilizaremos para poder denotar el tipo de cada expresión. De esta forma, podremos efectuar las distintas validaciones sobre los mismos.

```
⟨ TypeSystem ⟩ → int | real | bool | char | ⟨ tname ⟩
               | pointer ⟨ TypeSystem ⟩
               | array [⟨ ArrayIndex ⟩] of ⟨ TypeSystem ⟩
               | tuple [⟨ TupleField ⟩]
               | ⟨ typevar ⟩

⟨ ArrayIndex ⟩ → int | char | ⟨ tname ⟩ | ns

⟨ TupleField ⟩ → ⟨ fname ⟩ . ⟨ TypeSystem ⟩
```

Ahora ya estamos en condiciones para listar las distintas reglas de inferencia para expresiones. Utilizaremos la siguiente notación para especificar que la expresión e posee el tipo inferido θ bajo el contexto π.

$$\pi \vdash_e e : \theta$$

Para las distintas clases de constantes del lenguaje no se presenta ningún caso complejo. Notar que la constante *null* posee tipo polimórfico. Se listan a continuación.

Regla TC: Enteros

$$\frac{}{\pi \vdash_e n : int} \quad \text{cuando } n \in \langle int \rangle$$

Regla TC: Reales

$$\frac{}{\pi \vdash_e r : real} \quad \text{cuando } r \in \langle real \rangle$$

Regla TC: Booleanos

$$\frac{}{\pi \vdash_e b : bool} \quad \text{cuando } b \in \langle bool \rangle$$

Regla TC: Caracteres

$$\frac{}{\pi \vdash_e c : char} \quad \text{cuando } c \in \langle char \rangle$$

Regla TC: Constantes

$$\frac{\exists A.(t, A) \in \pi_{enum} \wedge e \in A}{\pi \vdash_e e : t} \quad \text{cuando } e \in \langle cname \rangle$$

Regla TC: Infinito

$$\frac{}{\pi \vdash_e inf : int}$$

Regla TC: Puntero Nulo

$$\frac{}{\pi \vdash_e null : pointer\ \theta} \quad \text{con } \theta \in \langle TypeSystem \rangle$$

En el caso de las variables hay tres reglas diferentes para la deducción de tipo. En base al contexto en el que fueron introducidas, se tendrá que emplear una u otra de las siguientes.

Regla TC: Variables Declaradas

$$\frac{(x, \theta) \in \pi_{var}}{\pi \vdash_e x : \theta}$$

Regla TC: Variables de Función

$$\frac{(f, \{(a_1, \theta_1), \ldots, (a_l, \theta_l)\}, (a_r, \theta_r)) \in \pi_{fun}}{\pi \vdash_e a_i : \theta_i}$$

durante el análisis del bloque $block_f$.

Regla TC: Variables de Procedimiento

$$\frac{(p, \{(oi_1, a_1, \theta_1), \ldots, (oi_l, a_l, \theta_l)\}) \in \pi_{proc}}{\pi \vdash_e a_i : \theta_i}$$

durante el análisis del bloque $block_p$.

Para los tipos estructurados se emplean las siguientes reglas. En el caso de los arreglos, el tipo de las expresiones debe coincidir con el de los índices especificados en la declaración del mismo. No se hace ninguna clase de chequeo para comprobar que el acceso sea dentro de los límites válidos.

Regla TC: Acceso a Puntero

$$\frac{\pi \vdash_e v : pointer\ \theta}{\pi \vdash_e \#v : \theta}$$

Regla TC: Acceso a Tuplas

$$\frac{\pi \vdash_e v : tuple\ f_1.\theta_1, \ldots, f_m.\theta_m}{\pi \vdash_e v.f_i : \theta_i}$$

Regla TC: Acceso a Arreglos

$$\frac{\pi \vdash_e v : array\ \theta_1, \ldots, \theta_n\ of\ \theta \qquad \pi \vdash_e e_1 : \theta_1 \qquad \ldots \qquad \pi \vdash_e e_n : \theta_n}{\pi \vdash_e v[e_1, \ldots, e_n] : \theta}$$

Algunos de los operadores del lenguaje se encuentran sobrecargados. Esto quiere decir que pueden ser utilizados para operar con valores de tipos diferentes. En particular, los numéricos aceptan valores enteros como reales. Otra de las cuestiones pendientes a debatir, es para que clase de tipos estarán definidas las operaciones de orden e igualdad. Nuevamente, esta es otra discusión sobre *typeclasses* y su implementación.

Regla TC: Operadores Binarios Numéricos

$$\frac{\pi \vdash_e e_1 : int \qquad \pi \vdash_e e_2 : int}{\pi \vdash_e e_1 \oplus e_2 : int} \qquad \frac{\pi \vdash_e e_1 : real \qquad \pi \vdash_e e_2 : real}{\pi \vdash_e e_1 \oplus e_2 : real}$$

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

Regla TC: Operadores Binarios Booleanos

$$\frac{\pi \vdash_e e_1 : bool \qquad \pi \vdash_e e_2 : bool}{\pi \vdash_e e_1 \otimes e_2 : bool}$$

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

Regla TC: Operadores Unarios Numéricos

$$\frac{\pi \vdash_e e : int}{\pi \vdash_e -e : int} \qquad \frac{\pi \vdash_e e : real}{\pi \vdash_e -e : real}$$

Regla TC: Operadores Unarios Booleanos

$$\frac{\pi \vdash_e e : bool}{\pi \vdash_e !e : bool}$$

Regla TC: Operadores de Igualdad y Orden

$$\frac{\pi \vdash_e e_1 : \theta \qquad \pi \vdash_e e_2 : \theta}{\pi \vdash_e e_1 \odot e_2 : bool}$$

donde $\odot \in \{<, >, <=, >=, ==, !=\}$.

Para terminar con las reglas de inferencia de expresiones, solo resta definir la adecuada para las llamadas a función. A continuación se detalla la misma.

Regla TC: Funciones

$$\frac{\pi \vdash_e e_1 : \theta_1 \qquad \ldots \qquad \pi \vdash_e e_n : \theta_n}{\pi \vdash_e f(e_1, \ldots, e_n) : \theta_r}$$

donde se satisface que $(f, \{(a_1, \theta_1), \ldots, (a_n, \theta_n)\}, (a_r, \theta_r)) \in \pi_{fun}$.

Lo último que resta por definir para finalizar los chequeos estáticos es el subtipado. El mismo simplemente permite convertir un valor de tipo *int* a uno de tipo *real*. Se detalla a continuación.

Regla TC: Subtipado

$$\frac{\pi \vdash_e e : int}{\pi \vdash_e e : real}$$