Semántica (Borrador)

Matias Gobbi

19 de marzo 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 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, 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. Su composición se describe a continuación.

$$\langle expr \rangle \rightarrow \langle const \rangle \mid \langle fcall \rangle \mid \langle op \rangle \mid \langle var \rangle$$

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.

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

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.

$$\langle fcall \rangle \rightarrow \langle id \rangle \ (\ \langle expr \rangle \ \ldots \ \langle expr \rangle \ )$$

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.

$$\langle op \rangle \rightarrow \langle expr \rangle \ \langle bin \rangle \ \langle expr \rangle \mid \langle un \rangle \ \langle expr \rangle$$

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

$$\langle un \rangle \rightarrow - \mid !$$

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.

$$\begin{aligned} \langle var \rangle \rightarrow & \ \langle id \rangle \\ & \mid \langle var \rangle \ [\ \langle expr \rangle \ \ldots \ \langle expr \rangle \ ] \\ & \mid \langle var \rangle \ . \ \langle fname \rangle \\ & \mid \# \ \langle var \rangle \end{aligned}$$

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.

$\langle sent \rangle \rightarrow$ **skip** | $\langle assign \rangle$ | $\langle pcall \rangle$ | $\langle if \rangle$ | $\langle while \rangle$ | $\langle for \rangle$

$\langle sblock \rangle \rightarrow \langle sent \rangle \ \ldots \ \langle sent \rangle$

$\langle assign \rangle \rightarrow \langle var \rangle := \langle expr \rangle$

$\langle while \rangle \rightarrow$ **while** $\langle expr \rangle$ **do** $\langle sblock \rangle$

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.

$\langle pcall \rangle \rightarrow \langle id \rangle$ ($\langle expr \rangle \ \ldots \ \langle expr \rangle$) | **alloc** $\langle var \rangle$ | **free** $\langle var \rangle$

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

$\langle if \rangle \rightarrow$ **if** $\langle expr \rangle$ **then** $\langle sblock \rangle \ \langle elif \rangle \ \ldots \ \langle elif \rangle \ \langle else \rangle$

$\langle elif \rangle \rightarrow$ **elif** $\langle expr \rangle$ **then** $\langle sblock \rangle$

$\langle else \rangle \rightarrow$ **else** $\langle sblock \rangle$ | ϵ

Finalmente, otra instrucción que presenta varias opciones es el *for*. Además de especificar rangos ascendentes o descendentes para la iteración, también se pueden detallar estructuras iterables.

$\langle for \rangle \rightarrow$ **for** $\langle id \rangle := \langle expr \rangle$ **to** $\langle expr \rangle$ **do** $\langle sblock \rangle$
| **for** $\langle id \rangle := \langle expr \rangle$ **downto** $\langle expr \rangle$ **do** $\langle sblock \rangle$
| **for** $\langle id \rangle$ **in** $\langle expr \rangle$ **do** $\langle sblock \rangle$

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.

$\langle type \rangle \rightarrow$ **int** | **real** | **bool** | **char**
| $\langle array \rangle$
| $\langle pointer \rangle$
| $\langle typevar \rangle$
| $\langle typedef \rangle$

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 tamaños de los mismos. El no terminal *bsize* representa al tamaño variable.

```
⟨ array ⟩ → array ⟨ size ⟩ ... ⟨ size ⟩ of ⟨ type ⟩

⟨ size ⟩ → ⟨ nat ⟩ | ⟨ bsize ⟩

⟨ 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 ⟩ ... ⟨ 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.

```
⟨ 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 métodos. A su vez, un bloque está conformado por una lista de declaraciones de variables acompañadas por una lista de sentencias.

Definir sintaxis para Instancias!

```
⟨ prog ⟩ → ⟨ typedecl ⟩ ... ⟨ typedecl ⟩ ⟨ methdecl ⟩ ... ⟨ methdecl ⟩

⟨ block ⟩ → ⟨ vardecl ⟩ ... ⟨ vardecl ⟩ ⟨ sblock ⟩

⟨ vardecl ⟩ → var ⟨ id ⟩ ... ⟨ id ⟩ : ⟨ type ⟩
```

Un método puede ser una función o un procedimiento. Ambos poseen un identificador propio, una lista de argumentos, y un bloque de instrucciones que conforman su cuerpo. También se pueden agregar restricciones para las variables de tipo en su prototipo.

```
⟨ methdecl ⟩ → ⟨ fun ⟩ | ⟨ proc ⟩

⟨ fun ⟩ → fun ⟨ id ⟩ ( ⟨ funarg ⟩ ... ⟨ funarg ⟩ ) ret ⟨ funret ⟩
          where ⟨ constraint ⟩ ... ⟨ constraint ⟩
          ⟨ block ⟩

⟨ funarg ⟩ → ⟨ id ⟩ : ⟨ type ⟩

⟨ funret ⟩ → ⟨ id ⟩ : ⟨ type ⟩

⟨ proc ⟩ → proc ⟨ id ⟩ ( ⟨ procarg ⟩ ... ⟨ procarg ⟩ )
           where ⟨ constraint ⟩ ... ⟨ constraint ⟩
           ⟨ block ⟩

⟨ procarg ⟩ → ⟨ io ⟩ ⟨ id ⟩ : ⟨ type ⟩

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

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 P posee la siguiente forma, donde $n \geq 0$ y $m > 0$.

$$
\begin{gathered}
typedecl_1 \\
\dots \\
typedecl_n \\
methoddecl_1 \\
\dots \\
methoddecl_m
\end{gathered}
$$

Ignoro la declaración de instancias!

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.

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

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. Validaciones en 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 en base a si se desea definir un tipo enumerado, un sinónimo de tipo, o una tupla.

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

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.

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

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. 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

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

Regla DT para Tipos: Punteros

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

Regla DT para Tipos: Arreglos

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

Regla DT para Tipos: Variables de Tipo

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

Regla DT para Tipos: Tipos Definidos

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

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

Para el caso de los tamaños de un arreglo, solo hay que verificar que los mismos sean naturales. Esto se debe a que en esta etapa, no se permiten tamaños variables para los mismos.

Regla DT para Tipos: Tamaños

$$\frac{}{\pi_{type} \vdash_s size} \quad \text{cuando } size \in \langle nat \rangle$$

3.1.2. Tipos 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.

$$FT : \langle type \rangle \to \{\ \langle typevar \rangle\ \}$$

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.

$$\begin{array}{lll}
FT(\textbf{int}) & = \emptyset & \\
FT(\textbf{real}) & = \emptyset & \\
FT(\textbf{bool}) & = \emptyset & \\
FT(\textbf{char}) & = \emptyset & \\
FT(\textbf{pointer } \theta) & = FT(\theta) & \\
FT(\textbf{array } s_1, \ldots, s_n \textbf{ of } \theta) & = FT(\theta) & \\
FT(vt) & = \{vt\} & vt \in \langle typevar \rangle \\
FT(t \textbf{ of } \theta_1, \ldots, \theta_n) & = FT(\theta_1) \cup \ldots \cup FT(\theta_n) &
\end{array}$$

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. En caso contrario, se debería detener el análisis del programa y generar un mensaje de error adecuado al conflicto encontrado durante la verificación. Notar que un tipo definido solo será accesible para las declaraciones posteriores.

Regla DT: Enumerados

$$\frac{\forall i \in \{1 \dots n\}, (t^*, A) \in \pi_{enum}.\ c_i \notin A}{\pi_{type} \vdash_{td} \textbf{enum}\ t = c_1, \dots, c_n}$$

donde todas las constantes c_i son distintas entre sí.

Regla DT: Sinónimos

$$\frac{\pi_{type} \vdash_t \theta}{\pi_{type} \vdash_{td} \textbf{syn}\ t\ \textbf{of}\ a_1, \dots, a_l = \theta}$$

donde se satisface que $\{a_1, \dots, a_l\} = FT(\theta)$, y todos los argumentos a_i son distintos entre sí.

Regla DT: Tuplas

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

donde se satisface que $\{a_1, \dots, a_l\} = FT(\theta_1) \cup \dots \cup FT(\theta_m)$, 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 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 la regla anterior para permitir esta situación obtenemos lo siguiente.

Regla DT: Recursión

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

donde se satisface que $\{a_1, \dots, a_l\} = FT(\theta_1) \cup \dots \cup FT(\theta_m)$, todos los alias f_i son distintos entre sí, y los argumentos a_i también.

3.2. Validaciones en 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.

$$methoddecl_i = \begin{cases} \textbf{fun } f_i\ (a_1 : \theta_1, \ldots, a_l : \theta_l) \textbf{ ret } a_r : \theta_r \\ \textbf{where } c_1\ \ldots\ c_m \\ \quad block_{f_i} \\ \textbf{proc } p_i\ (io_1\ a_1 : \theta_1, \ldots, io_l\ a_l : \theta_l) \\ \textbf{where } c_1\ \ldots\ c_m \\ \quad block_{p_i} \end{cases} \qquad \forall i \in \{1 \ldots m\}$$

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.

$$\pi_{fun} = \{(f, A, r, T) \mid$$
$$f \in \langle id \rangle \wedge A \subset \langle id \rangle \times \langle type \rangle \wedge r \in \langle id \rangle \times \langle type \rangle \wedge T \subset \langle typevar \rangle \times \langle class \rangle^+\}$$
$$\pi_{proc} = \{(p, A, T) \mid$$
$$p \in \langle id \rangle \wedge A \subset \langle io \rangle \times \langle id \rangle \times \langle type \rangle \wedge T \subset \langle typevar \rangle \times \langle class \rangle^+\}$$

Antes de comenzar con la validación de los métodos, es necesario definir dos 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.

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

En estos últimos contextos, se utiliza π_{loc} para almacenar la información relacionada con las variables declaradas en el cuerpo del método. Mientras que se usa π_{size} para guardar los identificadores para los tamaños variables de arreglos. Por último, con $\pi_{typevar}$ se busca recordar todos los tipos polimórficos introducidos en el prototipo del método. A continuación se detallan sus estructuras.

$$\pi_{loc} \subset \langle id \rangle \times \langle type \rangle$$
$$\pi_{size} \subset \langle bsize \rangle$$
$$\pi_{typevar} \subset \langle typevar \rangle$$

3.2.1. Prototipo de Métodos

Cuando se comienza con la verificación de un método, el contexto para variables debe estar vacío, al igual que el de variables de tipo. Los mismos se

iran completando a medida que avance el análisis del método. El contexto de tipos deberá estar previamente inicializado, con la información recolectada del análisis de declaración de tipos. El último contexto, el de métodos, es extendido solamente cuando se verifica por completo la validez de uno de estos.

¿Debería reflejar la modificación de contextos?

Regla M: Funciones

$$\frac{\underbrace{\pi \vdash_t \theta_1 \quad \ldots \quad \pi \vdash_t \theta_l}_{\text{argumentos}} \qquad \pi \vdash_t^* \theta_r \qquad \underbrace{\vdash_c c_1 \quad \ldots \quad \vdash_c c_m}_{\text{restricciones}} \qquad \pi^* \vdash_{mb} block_f}{\pi_0 \vdash_m \textbf{fun}\ f\ (a_1 : \theta_1, \ldots, a_l : \theta_l)\ \textbf{ret}\ a_r : \theta_r\ \textbf{where}\ c_1, \ldots, c_m\ block_f}$$

donde todos los identificadores a_i son distintos entre sí, y de los tamaños variables especificados en los tipos θ_j. Además, vale que $l \geq 0$.

Hay que hacer una salvedad para el análisis del retorno de la función. Ya que en el mismo no se pueden introducir nuevas variables de tipo o tamaños de arreglos variables. El conjunto de reglas que se utilizarán para deducir su tipo en el sistema de tipos será el empleado para la verificación de tipos en declaración de variables dentro de bloques.

Regla M: Procedimientos

$$\frac{\underbrace{\pi \vdash_t \theta_1 \quad \ldots \quad \pi \vdash_t \theta_l}_{\text{argumentos}} \qquad \underbrace{\vdash_c c_1 \quad \ldots \quad \vdash_c c_m}_{\text{restricciones}} \qquad \pi^* \vdash_{mb} block_p}{\pi_0 \vdash_m \textbf{proc}\ p\ (oi_1\ a_1 : \theta_1, \ldots, oi_l\ a_l : \theta_l)\ \textbf{where}\ c_1, \ldots, c_m\ block_p}$$

donde todos los identificadores a_i son distintos entre sí, y de los tamaños variables especificados en los tipos θ_j. Además, vale que $l > 0$.

Debido a que la utilidad principal de un procedimiento es poder modificar el valor de un conjunto de variables cuando es invocado, carecería de sentido permitir la especificación de uno que no altere ninguna de sus entradas. Por lo tanto, se debería satisfacer la siguiente ecuación para todas las declaraciones de procedimientos.

$$\exists i \in \{1 \ldots l\}.io_i = \textbf{out} \vee io_i = \textbf{in/out}$$

En el caso que no se satisfaga lo anterior, se debería generar una advertencia para informar al programador sobre un posible error en su código. La ejecución de un procedimiento que no cumpla la propiedad previa no produciría ningún efecto secundario en el entorno donde fue invocado.

3.2.2. Contextos Aumentados

Antes de analizar el cuerpo de una declaración de método, es necesario modificar el contexto actual para facilitar ciertas verificaciones. En definitiva, queremos permitir las llamadas recursivas. Para lograr esto, es imprescindible aumentar los contextos con la información del prototipo de la función o procedimiento correspondiente al bloque en cuestión. Por lo tanto, en las reglas anteriores se modifica π para poder analizar el cuerpo del método con el contexto π^*.

$$\pi_r^* = \pi_r \cup \{\ (f, \{(a_1, \theta_1), \ldots, (a_l, \theta_l)\}, (a_r, \theta_r), \ldots$$
$$\ldots \{(tv_1, \{c_1^1, \ldots c_r^1\}), \ldots, (tv_t, \{c_1^t, \ldots, c_r^t\})\})\ \}$$
$$\pi_l^* = \pi_l \cup \{\ (p, \{(oi_1, a_1, \theta_1), \ldots, (oi_l, a_l, \theta_l)\}, \ldots$$
$$\ldots \{(tv_1, \{c_1^1, \ldots c_r^1\}), \ldots, (tv_t, \{c_1^t, \ldots, c_r^t\})\})\ \}$$

Los valores de los elementos tv_i y c_j serán explicados en la sección de análisis en restricciones de métodos.

3.2.3. 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 previamente. Por lo cual, se necesita de otro conjunto de reglas para la validación de los mismos. En el prototipo de una función o procedimiento, se pueden especificar variables de tipo y tamaños variables de arreglos. 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. A continuación se detalla el conjunto de reglas de inferencia para los tipos especificados en los prototipos de métodos.

Regla M para Tipos: Básicos

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

Regla M para Tipos: Punteros

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

Regla M para Tipos: Arreglos

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

Regla M para Tipos: Variables de Tipo

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

además, se agrega la variable θ al contexto $\pi_{typevar}$.

Regla M para Tipos: Tipos Definidos

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

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

Las reglas para tamaños de arreglos son similares a las especificadas previamente, con la excepción que ahora se permiten usar tamaños variables. Por lo tanto, se agrega una nueva regla para la deducción.

Regla M para Tipos: Tamaños Constantes

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

Regla M para Tipos: Tamaños Variables

$$\frac{}{\pi \vdash_s size} \quad \text{cuando } size \in \langle bsize \rangle$$

además, se agrega la variable $size$ al contexto π_{size}.

¿Debería agregar alguna premisa adicional a la regla?

3.2.4. Restricciones de Métodos

Algunos métodos presentan ciertas restricciones para las variables de tipo especificadas en su prototipo. Básicamente, se imponen clases que deben ser implementadas por los tipos en los que se evaluarán las variables.

Regla M: Restricciones

$$\frac{}{tv : c_1, \ldots, c_m}$$

donde todas las clases c_i son distintas entre sí. Además, se satisface que $tv \in \pi_{typevar}$. Finalmente, se agrega el par $(tv, \{c_1, \ldots, c_m\})$ al contexto del prototipo del método correspondiente.

En el caso que se agregara un par $(tv, \{c_1, \ldots, c_m\})$ donde la variable tv ya estuviese presente en el conjunto T, del contexto (f, A, r, T) o (p, A, T), se debería generar un mensaje de error debido a que hay dos restricciones distintas para la misma variable de tipo.

3.3. Validaciones en 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 del método en cuestión, y obtienen el conjunto de identificadores especificados en el encabezado del mismo.

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

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

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.

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

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

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 del método analizado. Notar que la estructura de este conjunto es independiente a la clase de método que se esté analizando.

$$Vars_{\pi_{var}} = \{v \mid \exists \theta.(v, \theta) \in \pi_{loc}\} \cup \pi_{size}$$

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) del método en cuestión, de los tamaños 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.

$$NameSpace = "identificadores\ en\ uso\ en\ el\ alcance\ actual" \subset \langle id \rangle$$
$$NameSpace_f = Argumentos(f) \cup Retorno(f) \cup Vars$$
$$NameSpace_p = Inputs(p) \cup Ouputs(p) \cup InOuts(p) \cup Vars$$

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

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

donde todos los identificadores x_i son distintos entre sí. Además, se deberán agregar todos los pares (x_i, θ) al contexto π_{loc}.

3.4.1. 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 exclusivamente a las introducidas en el prototipo del método analizado. Para los tamaños variables debemos hacer algo similar. La mayoría de las reglas permanecen sin cambios.

Regla B para Tipos: Básicos

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

Regla B para Tipos: Punteros

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

Regla B para Tipos: Arreglos

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

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

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

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

Las reglas para la especificación de rangos son similares a las anteriores. Solo hay dos casos. El primero es trivial, corresponde a la utilización de tamaños constantes, por lo que se procede de la misma manera que se hizo anteriormente. El segundo refiere al empleo de tamaños variables, donde solo hay que verificar que el identificador haya sido introducido previamente.

Regla B para Tipos: Tamaños Constantes

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

Regla B para Tipos: Tamaños Variables

$$\frac{size \in \pi_{size}}{\pi \vdash_s size} \quad \text{cuando } size \in \langle bsize \rangle$$

3.5. 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.

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

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.

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

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.

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

¿Alguna forma de analizar estáticamente si se modifican punteros?

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*.

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

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

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.

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

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.

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

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

Determinar cuando hay pasaje por valor y cuando por referencia.

3.6. Validaciones en 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.

¿Debería comprobar si las restricciones de clases son efectivamente necesarias?

Función Phi: Uso efectivo de variables.

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

Función Phi: Empleo de variables.

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

Función Phi: Evitar modificación de tamaños.

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

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

Función Phi: Evitar modificación de argumentos.

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

Función Phi: 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 Phi: Etiquetas *in* respetadas.

$$Inputs(\gamma) \cap AV(sblock) = \emptyset \quad \text{lo cual implica} \quad Inputs(\gamma) \subset RV(sblock)$$

Función Phi: Etiquetas *out* respetadas.

$$Outputs(\gamma) \cap RV(sblock) = \emptyset \quad \text{lo cual implica} \quad Outputs(\gamma) \subset AV(sblock)$$

Función Phi: Etiquetas *in/out* respetadas.

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

Verificar si una variable fue asignada previa a su lectura debe ser dinámica. Lo mismo con punteros.

Ahora definiremos propiamente las reglas de inferencia para las sentencias. Las mismas no ofrecen 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

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

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 necesitan realizar un par de verificaciones adicionales. La primera tiene que ver con el polimorfismo que admite el método y como chequear su consistencia.

$$\_ \mid \_ : \langle type \rangle \times \Delta \to \langle type \rangle$$
$$\Delta_{bs} = \langle bsize \rangle \to \langle size \rangle$$
$$\Delta_{tv} = \langle typevar \rangle \to \langle type \rangle$$

Es necesario definir una función de sustitución para poder verificar si los tipos de las expresiones en los argumentos coinciden con el de las entradas actuales del procedimiento. La función de sustitución tendría la siguiente forma.

$$\begin{aligned}
&\textbf{int} \mid \delta && = \textbf{int} \\
&\textbf{real} \mid \delta && = \textbf{real} \\
&\textbf{bool} \mid \delta && = \textbf{bool} \\
&\textbf{char} \mid \delta && = \textbf{char} \\
&\textbf{pointer}\ \theta \mid \delta && = \textbf{pointer}\ (\theta \mid \delta) \\
&\textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ \theta \mid \delta && = \textbf{array}\ (s_1 \mid \delta), \ldots, (s_n \mid \delta)\ \textbf{of}\ (\theta \mid \delta) \\
&vt \mid \delta && = \delta_{tv}(vt) \qquad vt \in \langle typevar \rangle \\
&t\ \textbf{of}\ \theta_1, \ldots, \theta_n \mid \delta && = t\ \textbf{of}\ (\theta_1 \mid \delta), \ldots, (\theta_n \mid \delta)
\end{aligned}$$

La sustitución se tiene que propagar también para los tamaños de arreglos. Esto se debe al uso de tamaños variables para especificar los mismos. Para los rangos, la función tendría el siguiente comportamiento.

$$\begin{array}{lll} size \mid \delta & = size & size \in \langle nat \rangle \\ size \mid \delta & = \delta_{bs}(size) & size \in \langle bsize \rangle \end{array}$$

La segunda verificación, está relacionada con la forma de los argumentos en su llamada. 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.

¿Esta bien sustituir también los tamaños variables? Suena muy restrictivo. PD: Ya me dieron el OK

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 el procedimiento se encuentra declarado con la siguiente información.

$$(p, \{(io_1, a_1, \theta_1^*), \ldots, (io_n, a_n, \theta_n^*)\}, T) \in \pi_{proc}$$

Los tipos de los argumentos actuales y reales se igualan, luego de sustituir.

$$\exists \delta \in \Delta. \forall i \in \{1 \ldots n\}.\ (\theta_i^* \mid \delta = \theta_i)$$

Las entradas de escritura son efectivamente variables modificables.

$$\forall i \in \{1 \ldots n\}.\ a_i \in Outputs(p) \cup InOuts(p) \implies e_i \in \langle var \rangle$$

Todos los tipos satisfacen sus respectivas restricciones.

$$\forall (tv, \{c_1, \ldots, c_m\}) \in T.tv \mid \delta = \theta \implies \theta \text{ satisface las clases } c_1, \ldots, c_m$$

Regla B: Alloc

$$\frac{\pi \vdash_e v : \textbf{pointer}\ \theta}{\pi \vdash_s \textbf{alloc}\ v} \quad \text{con } \theta \in \langle type \rangle$$

Regla B: Free

$$\frac{\pi \vdash_e v : \textbf{pointer}\ \theta}{\pi \vdash_s \textbf{free}\ v} \quad \text{con } \theta \in \langle type \rangle$$

Regla B: While

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

Regla B: If

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

Finalmente, especificamos las reglas para las sentencias *for*. Todavía queda pendiente definir como se implementarán las *typeclasses* en el lenguaje. En primera instancia, los enteros, los caracteres, y los enumerados deberían ser enumerables, mientras que un arreglo en sí sería iterable. En las sucesivas fases de desarrollo del intérprete se tomará una decisión respecto al asunto. Debido a esto, las siguientes pruebas pueden resultar ambiguas.

Regla B: For To

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

donde se satisface que el tipo θ es enumerable.

Regla B: For Downto

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

donde se satisface que el tipo θ es enumerable.

Regla B: For In

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

donde se satisface que el tipo θ es iterable.

3.7. Validaciones en 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 junto con las reglas de subtipado y unificación apropiadas. 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 el caso de las constantes del lenguaje, no se presenta ninguna situación compleja. Las reglas son directas. Notar que la constante *null* posee tipo polimórfico, mientras que *inf* posee tipo entero. A continuación se listan las reglas de inferencia.

Regla TC: Valores Constantes

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

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

Regla TC: Constantes Enumeradas

$$\frac{}{\pi \vdash_e e : t} \quad \text{cuando } e \in \langle cname \rangle$$

donde se satisface que $\exists A.(t, A) \in \pi_{enum} \wedge e \in A$.

Regla TC: Infinito

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

Regla TC: Puntero Nulo

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

En el caso de las variables 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 deducciones.

Regla TC: Variables Declaradas

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

Regla TC: Tamaños Variables

$$\frac{x \in \pi_{size}}{\pi \vdash_e x : \textbf{int}}$$

Regla TC: Variables de Función

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

durante el análisis del bloque $block_f$.

Regla TC: Variables de Procedimiento

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

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 ser entero. No se hace ninguna clase de chequeo para comprobar que el acceso a una de estas estructuras sea dentro de los límites válidos. Esta verificación es delegada al análisis dinámico para realizar durante la ejecución del programa.

Regla TC: Acceso a Puntero

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

Regla TC: Acceso a Tuplas

$$\frac{\pi \vdash_e v : t\ \textbf{of}\ \theta_1, \ldots, \theta_l}{\pi \vdash_e v.f_i : \theta_i^* \mid [a_1 : \theta_1, \ldots, a_l : \theta_l]}$$

donde se satisface que $(t, \{a_1, \ldots, a_l\}, \{(f_1, \theta_1^*), \ldots, (f_m, \theta_m^*)\}) \in \pi_{tuple}$.

Regla TC: Acceso a Arreglos

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

Varios de los operadores del lenguaje 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. En particular, los numéricos aceptan valores enteros como reales.

Regla TC: Operadores Binarios Numéricos

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

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

Regla TC: Operadores Binarios Booleanos

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

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

Regla TC: Operadores Unarios Numéricos

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

Regla TC: Operadores Unarios Booleanos

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

Otra de las cuestiones pendientes a debatir en el lenguaje, es para que clase de tipos estarán definidas las operaciones de orden e igualdad. Nuevamente, esta es otra discusión sobre *typeclasses* y su implementación. Podríamos asumir que todos los tipos cumplen las condiciones para ser igualables. Mientras, solo los tipos básicos y los enumerados podrían ser ordenables.

Regla TC: Operadores de Igualdad

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

donde $\odot \in \{==, != \}$, y el tipo θ es igualable.

Regla TC: Operadores de Orden

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

donde $\odot \in \{<, >, <=, >=\}$, y el tipo θ es ordenable.

Para terminar con las reglas de inferencia de expresiones, solo resta definir la adecuada para las llamadas a función. Similar a los procedimientos, se necesita comprobar la adecuación de los tipos polimórficos empleados. 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 la función se encuentra declarada con la siguiente información.

$$(f, \{(a_1, \theta_1^*), \ldots, (a_n, \theta_n^*)\}, (a_r, \theta_r^*), T) \in \pi_{fun}$$

Los tipos de los argumentos actuales y reales se igualan, luego de sustituir.

$$\exists \delta \in \Delta.\ (\forall i \in \{1 \ldots n\}.\ \theta_i^* \mid \delta = \theta_i) \wedge (\theta_r^* \mid \delta = \theta_r)$$

Todos los tipos satisfacen sus respectivas restricciones.

$$\forall (tv, \{c_1, \ldots, c_m\}) \in T.tv \mid \delta = \theta \implies \theta \text{ satisface las clases } c_1, \ldots, c_m$$

Una regla fundamental para el tipado de las expresiones tiene que ver con el subtipado. 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

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

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. Aunque se representa solo con su nombre y sus argumentos, se puede establecer una igualdad semántica con el tipo en el cuerpo de su definición.

Antes de describir las reglas de unificación de tipos, debemos redefinir la función de sustitución anterior. En esta ocasión queremos ser capaces de reemplazar las variables de tipo en el cuerpo de una declaración de tipo, por los argumentos en los que se instanciará el mismo.

$$\_ \mid \_ : \langle type \rangle \times \Delta \to \langle type \rangle$$
$$\Delta = \langle typevar \rangle \to \langle type \rangle$$

La estructura de la función es casi idéntica a la que se dio previamente. La diferencia principal es que nos limitaremos a sustituir solo variables de tipo. Al no permitirse el uso de tamaños variables en declaraciones de tipos, no necesitamos preocuparnos por este aspecto.

$$\begin{aligned}
&\textbf{int} \mid \delta && = \textbf{int} \\
&\textbf{real} \mid \delta && = \textbf{real} \\
&\textbf{bool} \mid \delta && = \textbf{bool} \\
&\textbf{char} \mid \delta && = \textbf{char} \\
&\textbf{pointer}\ \theta \mid \delta && = \textbf{pointer}\ (\theta \mid \delta) \\
&\textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ \theta \mid \delta && = \textbf{array}\ s_1, \ldots, s_n\ \textbf{of}\ (\theta \mid \delta) \\
&vt \mid \delta && = \delta(vt) \qquad vt \in \langle typevar \rangle \\
&t\ \textbf{of}\ \theta_1, \ldots, \theta_n \mid \delta && = t\ \textbf{of}\ (\theta_1 \mid \delta), \ldots, (\theta_n \mid \delta)
\end{aligned}$$

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.

Regla U: Reflexividad

$$\frac{}{\theta \sim \theta}$$

Regla U: Simetría

$$\frac{\theta \sim \theta'}{\theta' \sim \theta}$$

Regla U: Transitividad

$$\frac{\theta \sim \theta' \qquad \theta' \sim \theta''}{\theta \sim \theta''}$$

Regla U: Unificación

$$\frac{\pi \vdash_e e : \theta \qquad \theta \sim \theta'}{\pi \vdash_e e : \theta'}$$

Regla U: Punteros

$$\frac{\theta \sim \theta'}{\textbf{pointer } \theta \sim \textbf{pointer } \theta'}$$

Regla U: Arreglos

$$\frac{\theta \sim \theta'}{\textbf{array } s_1, \ldots, s_n \textbf{ of } \theta \sim \textbf{array } s_1, \ldots, s_n \textbf{ of } \theta'}$$

Regla U: Tipos Definidos

$$\frac{\theta_1 \sim \theta'_1 \qquad \ldots \qquad \theta_n \sim \theta'_n}{t \textbf{ of } \theta_1, \ldots, \theta_n \sim t \textbf{ of } \theta'_1, \ldots, \theta'_n}$$

Regla U: Sinónimos

$$\frac{}{t \textbf{ of } \theta_1, \ldots, \theta_n \sim \theta \mid [a_1 : \theta_1, \ldots, a_n : \theta_n]}$$

donde se satisface que $(t, \{a_1, \ldots, a_n\}, \theta) \in \pi_{syn}$.

¿Son suficientes las reglas de unificación? Para mí, si.