Capítulo 1

Sintaxis y semántica (boceto)

1.1. Sintaxis de CABS (Still in development)

Por el momento véase la especificación en el repositorio del compilador. Para sentar las bases de la notación que se usarán de ahora en adelante para definir la semántica diremos que **P** es un programa de nuestro lenguaje CABS formado por instrucciones globales como definición de variables globales (del estilo t var; con tipo y nombre de variable) y definición de funciones.

Las funciones estarán formadas por un tipo, un nombre de función, una lista de argumentos (posiblemente vacia), un cuerpo con instrucciónes $S \in \mathbf{Stm}$ y una expresión de retorno. Todo sigue una sintaxis sencilla C-like.

Las instrucciones S de un programa son las típicas de un lenguaje imperativo con asignaciones, operaciones aritmetico lógicas, instrucciones de control y, la joya de la corona, un **thread** para la ejecución de funciones con concurrencia imitando el comportamiento del fork en C.

1.2. Semántica de CABS (faltan arrays!!)

A continuación pasamos a hablar de la semántica del lenguaje CABS.

1.2.1. Preámbulo semántico

Antes de hablar de la semántica propiamente dicha necesitamos definir unas estructuras que nos serán posteriormente de gran utilidad.

En primer lugar definimos el conjunto de valores $\mathbb{V} = \mathbb{Z} \cup \mathbb{B}$ como la unión de los enteros y de los booleanos.

Una primera parte de la representación del estado estará formada por las variables globales de nuestro programa. Definimos $\mathbf{G} = \mathbf{Var} \hookrightarrow \mathbb{V}$ el conjunto de funciones parciales del conjunto de variables al conjunto de valores. El estado actual de nuestras variables globales será una función de este conjunto y por lo general nos referiremos a ella con la letra G. Posteriormente cuando hablemos de variables locales también las definiremos como un conjunto de funciones similares pero que trataremos por separado por comodidad.

Por otro lado nos encontramos en nuestro lenguaje con la necesidad de definir un conjunto que recoja la información básica de las funciones y procedimientos. Definimos $\mathbf{F} = \mathbf{Func} \hookrightarrow (\mathbf{T} \times \mathbf{Stm} \times \mathbf{Args} \times (\mathbf{Exp} \cup \{\varepsilon\}))$ el conjunto de funciones parciales que asocia un nombre de función a su definición. Una función quedará definida por su tipo de retorno $t \in \mathbf{T}$, su código $S \in \mathbf{Stm}$, sus argumentos de entrada $\mathrm{arg} \in \mathbf{Args}$ y su expresión de retorno $e \in \mathbf{Exp} \cup \{\varepsilon\}$. Según las necesidades, la expresión de retorno será una expresión booleana, una expresión aritmética o simplemente será la expresión vacia, empleada para los procedimientos.

Para dar un significado a nuestro programas tenemos que definir qué va a representar para nosotros su estado de ejecución. Definimos el conjunto de estados $\mathbf{State} = \mathbf{G} \times \mathbf{F} \times \mathbf{RP}$ como una tupla que recoja la información de las variables globales y de las funciones definidas en nuestro programa $\mathbf{P}$ así como una pila de marcos de ejecución o runtime processes que contendrá la información local de los procesos (e.g. variables locales (¡pilas de ámbitos locales!), código y tipo (¿y expresión de retorno?)).

La idea a seguir para definir nuestra semántica será apoyarnos en dos funciones auxiliares **init** y **start** que respectivamente inicializarán el estado global del programa y lanzarán a ejecución la función inicial *main*. Pasemos pues a definir la primera de ellas.

La función init.

Definimos la función $\mathbf{init} : \mathbf{Prog} \hookrightarrow \mathbf{State}$ de forma recursiva del siguiente modo.

$$
\begin{aligned}
\mathbf{init}(\varepsilon) &= (\mathbf{nil}, \mathbf{nil}, []) \\
\mathbf{init}(\mathit{int}\ \mathrm{var};\ \mathbf{P}) &= (G\,[\ \mathrm{var} \mapsto 0]\,, F, RP) \text{ donde } \mathbf{init}(\mathbf{P}) = (\mathbf{G}, \mathbf{F}, \mathbf{RP}) \\
\mathbf{init}(\mathit{bool}\ \mathrm{var};\ \mathbf{P}) &= (G\,[\ \mathrm{var} \mapsto \textsc{false}]\,, F, RP) \text{ donde } \mathbf{init}(\mathbf{P}) = (\mathbf{G}, \mathbf{F}, \mathbf{RP}) \\
\mathbf{init}(\mathit{int}\,\mathbf{func}(\mathrm{arg})\{S;\ \mathbf{return}\ a\}\mathbf{P}) &= (G, F\,[\mathbf{func} \mapsto (\mathit{int}, S, \mathrm{arg}, a)]\,, RP) \\
&\quad\ \text{donde } \mathbf{init}(\mathbf{P}) = (\mathbf{G}, \mathbf{F}, \mathbf{RP}) \\
\mathbf{init}(\mathit{bool}\,\mathbf{func}(\mathrm{arg})\{S;\ \mathbf{return}\ b\}\mathbf{P}) &= (G, F\,[\mathbf{func} \mapsto (\mathit{bool}, S, \mathrm{arg}, b)]\,, RP) \\
&\quad\ \text{donde } \mathbf{init}(\mathbf{P}) = (\mathbf{G}, \mathbf{F}, \mathbf{RP}) \\
\mathbf{init}(\mathit{void}\,\mathbf{func}(\mathrm{arg})\{S\}\mathbf{P}) &= (G, F\,[\mathbf{func} \mapsto (\mathit{void}, S, \mathrm{arg}, \varepsilon)]\,, RP) \\
&\quad\ \text{donde } \mathbf{init}(\mathbf{P}) = (\mathbf{G}, \mathbf{F}, \mathbf{RP})
\end{aligned}
$$

La función (regla) start.

Definimos la función $\mathbf{start} : \mathbf{State} \hookrightarrow \mathbf{State}$ como

$$
\mathbf{start}((\mathbf{G}, \mathbf{F}, \mathbf{RP})) = (\mathbf{G}, \mathbf{F}, [(\mathbf{nil} : [], S)]) \text{ donde } \mathbf{F}(\mathit{main}) = (\mathit{int}, S, \mathrm{arg}, 0)
$$

Con esto conseguimos crear un nuevo marco de ejecución con el código inicial de main. Nótese que la función inicializa la pila de ambitos de variables locales con el ambito **nil** que no contiene ninguna variable inicializada.

Semántica (Expresiones aritmético-lógicas).

(Primero hablemos de las funciones aritméticas que ya son un jaleo. Se trata de expresiones con operandos, numerales y variables junto con llamadas funciones (por el momento como si no tuvieran argumentos!!!)).

Tenemos que definir una función (parcial) que dada una expresión (aritmética reducida) nos devuelva otra expresión más simplificada, pudiendo emplear un estado para ello (permitiendo modificaciones del mismo), hasta eventualmente quedarnos con un valor (entero por el momento).

De ahora en adelante nos referimos por el conjunto **Aexp** a la unión de expresiones aritméticas y valores enteros.

Buscamos definir la función semántica $\mathcal{A} : (\mathbf{Aexp} \times \mathbf{State}) \hookrightarrow (\mathbf{Aexp} \times \mathbf{State})$ mediante las siguientes reglas:

$$[\text{num}_{\mathcal{A}}] \quad \frac{}{\langle n, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle \mathcal{N}\,[\![n]\!]\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle}$$

$$[\text{var}^{L}_{\mathcal{A}}] \quad \frac{local(x) = v \qquad G(x) = \text{ undef}}{\langle x, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Aexp} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle}$$

$$[\text{var}^{G}_{\mathcal{A}}] \quad \frac{G(x) = v \qquad local(x) = undef}{\langle x, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Aexp} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle}$$

$$[\bigodot^{1}_{\mathcal{A}}] \quad \frac{\langle a_1, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle a_1', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}{\langle a_1 \bigodot a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle a_1' \bigodot a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}$$

$$[\bigodot^{2}_{\mathcal{A}}] \quad \frac{\langle a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle a_2', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}{\langle v \bigodot a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle v \bigodot a_2', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}$$

$$[\bigodot^{3}_{\mathcal{A}}] \quad \frac{}{\langle v_1 \bigodot v_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle v_1 \bigodot_{\mathcal{N}} v_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle}$$

$$[\text{unstack}^{1}_{\mathcal{A}}] \quad \frac{\langle a, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle a', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}{\langle \textsc{unstack}\,(a)\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Aexp} \langle \textsc{unstack}\,(a')\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}$$

$$[\text{unstack}^{2}_{\mathcal{A}}] \quad \frac{}{\langle \textsc{unstack}\,(v)\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S)))\rangle \rightarrow_{Aexp} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle}$$

$$[\text{call}_{\mathcal{A}}]\ \frac{\mathbf{F}(\mathbf{func}) = (int, S_F, [], a)}{\langle \mathbf{func}(), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))) \rangle \rightarrow_{Aexp} \langle \text{UNSTACK}\,(a)\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathbf{nil} : local : s, S_F;\ S))) \rangle}$$

Semántica (Instrucciones). (Hay que meter más tipos, limitar declaraciones y un largo etc)

$$\left[\text{Decl}_{\text{C}}^{int}\right]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, int \text{ var}; S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local\,[\ \text{var} \mapsto 0] : s, S)))}$$

$$\left[\text{Decl}_{\text{C}}^{bool}\right]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, bool \text{ var}; S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local\,[\ \text{var} \mapsto \text{FALSE}] : s, S)))}$$

$$[\text{ass}_{\text{C}}^{1}]\ \frac{\langle a, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \varepsilon))) \rangle \rightarrow_{Aexp} \langle a', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathcal{A}}))) \rangle}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s,\ \text{var} = a;\ S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathcal{A}};\ \ \text{var} = a';\ S)))}$$

$$[\text{ass}_{\text{C}}^{2}]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s,\ \text{var} = v;\ S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local\,[\ \text{var} \mapsto v] : s, S)))}$$

$$[\text{if}_{\text{C}}]\ \frac{\langle b, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \varepsilon))) \rangle \rightarrow_{Bexp} \langle b', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathbb{B}}))) \rangle}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathbb{B}};\ \mathbf{if}(b')\{S_1\}\mathbf{else}\{S_2\}S)))}$$

$$\left[\text{if}_{\text{C}}^{\text{TRUE}}\right]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{if}(\text{TRUE})\{S_1\}\mathbf{else}\{S_2\}S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S_1;\ S)))}$$

$$\left[\text{if}_{\text{C}}^{\text{FALSE}}\right]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{if}(\text{FALSE})\{S_1\}\mathbf{else}\{S_2\}S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S_2;\ S)))}$$

$$[\text{while}_{\text{C}}]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{while}(b)\{S_1\}S))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{if}(b)\{S_1;\ \mathbf{while}(b)\{S_1\}\}S)))}$$

$$[\text{end}_{\text{C}}]\ \frac{}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, \varepsilon))) \rightarrow (\mathbf{G}, \mathbf{F}, \mathbf{RP}))}$$

$$[\text{thread}_{\text{C}}]\ \frac{\mathbf{F}(\mathbf{func}) = (t, S_F, [], e)}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, \mathbf{thread\,func}();\ S))) \rightarrow (\mathbf{G}, \mathbf{F}, (\mathbf{RP} \cup (\mathbf{nil} : [], S_F)) \rightsquigarrow (s, S))}$$

Capítulo 2

Traducción a ABS

La traducción de CABS a ABS forma el grueso de este trabajo. La diferencia de paradigmas entre un lenguaje y otro hace que sea necesario la implementación de algunas estructuras de datos adicionales ausentes en ABS.

Además de dichas estructuras, es necesario emplear algunos trucos para poder vencer las restricciones del lenguaje para crear el concepto de variable global o de llamada a funciones síncrona.

En este capítulo discutiremos este tema de una forma abstracta para posteriomente poder llevar a cabo la implementación de un compilador correcto.

2.1. Variables globales y funciones

La ausencia de memoría compartida entre los distintos cogs hace que la idea de variable global no sea inmediata. Del mismo modo, es necesario discutir el concepto de función al estilo de C, pese a que los métodos de una interfaz en ABS sean públicos y a primera vista similares.

Una primera aproximación vendría dada por el uso de una única clase en la que encapsular todo nuestro programa. Esta clase implementaría una interfaz con todas las cabeceras de las funciones de nuestro programa. Además contaría entre sus atributos las variables globales, consiguiendo de este modo una visibilidad completa desde cualquier punto del programa.

Veamos que ocurre con el siguiente código de ejemplo en CABS. En él se puede ver como se llama a la función f con **thread** haciendo que las dos asignaciones de la variable $var1$ puedan entrelazarse.

```
int var1;

void f() {
    var1 = 2;
```

```
}

int main() {
    thread f();
    var1 = 1;
    return 0;
}
```

En otras palabras, queremos que nuestra traducción a ABS pueda llegar a ambos interleavings, es decir, que el valor final de *var*1 pueda ser 1 o 2.

Empleando la primera aproximación, obtendríamos un código similar al siguiente.

```
module Cabs;
import * from ABS.StdLib;

interface Functions {
  Unit f();
  Unit main();
}

class Prog() implements Functions {
  Int var1 = 0;

  Unit f() {
    var1 = 2;
  }

  Unit main() {
    this!f();
    var1 = 1;
  }
}

{
  Functions prog = new Prog();
  prog.main();
}
```

Sin embargo, esta traducción nos lleva a un código ABS donde solo es posible uno de los dos interleavings (concretamente el que termina con *var*1 valiendo 2).

El motivo por el que ocurre esto es por el concepto de cog discutido anteriormente. (AÑADIR INTRO ABS!!!) En este programa existe un único cog que se corresponde con la instancia *prog*. Cuando la función *main* llama asíncronamente a *f* se realiza un paso de mensaje al mismo objeto, que lo almacena en una cola a la espera de poder ejecutarlo, es decir, a que termine la ejecución del mensaje actual que se corresponde con la llamada a *main*.

Un intento para solucionar esta situación podría pasar por usar un *await* en la llamada a f.

```
    await this!f();
```

De este modo conseguiremos que el mensaje del *main* pueda dejar paso a otro mensaje de la cola como la llamada a f permitiendo los dos interleavings.

Pero, ¿qué ocurre si el cuerpo de f y *main* son un poco más largos como en el siguiente ejemplo?

```
int var1;
int var2;

void f() {
    var1 = 2;
    var2 = 4;
}

int main() {
    thread f();
    var1 = 1;
    var2 = 3;
    return 0;
}
```

En este caso no hay forma de regresar a la función *main* antes de que termine la ejecución de f suponiendo que se escoja tras el await. De nuevo solo conseguimos la mitad de los 4 estados finales posibles.

La única alternativa que nos queda es tener cada función en un cog distinto. Y realizar una instanciación cada vez que realicemos una llamada.

Esta situación resuelve los problemas anteriormente planteados porque cada cog actúa como si se ejecutara en un procesador independiente. Queda ahora resolver cómo conseguir que haya variables compartidas entre los distintos cogs.

La solución pasa por crear un cog independiente en el que se almacenen las variables globales como atributos a los que solo se pueda acceder a través de unos métodos getters y setters. Este cog sería el primero en crearse y se pasaría como parámetro en la creación de los sucesivos.

Las llamadas a los mencionados getters and setters podrían hacerse usando una llamada asíncrona e inmediatamente haciendo un *await* de esta. Este *await* sería necesario tanto para lecturas (como se vio en la discusión de los Future (!!!!!)) como en las escrituras, puesto que un mismo cog podría intentar hacer dos escrituras seguidas sobre la misma variable y el orden de ellas debe mantenerse.

Usando esta idea el último ejemplo se podría traducir al siguiente código ABS:

```
module Cabs;
import * from ABS.StdLib;

interface GLOBAL {
  Int getvar1();
  Unit setvar1(Int val);
  Int getvar2();
  Unit setvar2(Int val);
}

class GlobalVariables() implements GLOBAL {
  Int var1 = 0;
  Int var2 = 0;

  Int getvar1() {
    return var1;
  }

  Unit setvar1(Int val) {
   var1 = val;
  }

  Int getvar2() {
    return var2;
  }

  Unit setvar2(Int val) {
    var2 = val;
  }
}

interface Intf {
  Unit f();
}

interface Intmain {
  Int main();
}

class Impf(GLOBAL globalval) implements Intf {
  Unit f() {
    await globalval!setvar1(2);
    await globalval!setvar2(4);
  }
}
```

```

class Impmain(GLOBAL globalval) implements Intmain {
  Int main() {
    Intf aux_f = new Impf(globalval);
    aux_f!f();
    await globalval!setvar1(1);
    await globalval!setvar2(3);
    return 0;
  }
}

{
  GLOBAL globalval = new GlobalVariables();
  Intmain prog = new Impmain(globalval);
  prog.main();
}
```

Analicemos por qué esto funciona. La ejecución de f y del resto de la función *main* se ejecutan ahora en paralelo. Ambas funciones tienen acceso al objeto *globalval* por ser un parámetro de clase. Cuando f o *main* realizan una escritura sobre $var1$ llaman al método setter correspondiente. Puede ocurrir que uno de los dos cogs realice la llamada y obtenga el valor de retorno antes que el otro llame al método o que los dos llamen a la vez (relativamente) y hagan un await.

En este segundo caso hay que analizar que no haya problemas de deadlock. Los métodos del cog *globalval* no realizan ninguna llamada a ninguna otra función. Solamente cogen el valor solicitado y lo devuelven, en el caso de las lecturas, o escriben un valor en un atributo, en el caso de las escrituras. Cuando dos o más llamadas a un método ocurren a la vez estas se introducen en la cola de mensajes del cog y se gestionan arbitrariamente. De este modo se garantiza que las llamadas a *globalval* terminan en algún momento y la arbitrariedad en la gestión de las llamadas garantizan la existencia de todos los interleavings posibles sobre la variable $var1$.

Del mismo modo se procede con la variable $var2$. Y gracias al *await* dentro de un mismo cog (como en la función f) se garantiza que no hay un desorden entre la asignación a $var1$ y la asignación a $var2$. De forma análoga esto funcionaría con un ejemplo en el que se hicieran lecturas.

Con esto hemos conseguido solucionar el problema de las variables globales y a la vez hemos obtenido una concurrencia de grano fino entre las distintas hebras de CABS al separar las lecturas de la escrituras en dos llamadas asíncronas distintas.

2.2. Arrays locales y globales

ABS cuenta un tipo genérico de lista recursiva de coste de acceso lineal. Se trata de un tipo inmutable que no permite la modificación del contenido de la lista una vez creada.

Pese a tener estas restricciones podemos conseguir un comportamiento similar al de un array común obviando el coste de las operaciones.

Para ello nos interesa encapsular el tipo lista en una clase ABS que implemente una interfaz con dos métodos que aporten el concepto de array: un getter y un setter de elementos por posición.

La implementación de un array de enteros quedaría del siguiente modo:

```
interface ArrayInt {
  Int getV(Int indx);
  Unit setV(Int indx, Int value);
}

class ArrayIntC(Int size) implements ArrayInt{
  List<Int> list = copy(0, size);

  Int getV(Int indx) {
    return nth(this.list, indx);
  }

  Unit setV(Int indx, Int value) {
    Int i = 0;
    List<Int> prev = Nil;
    List<Int> post = list;
    while (i < indx) {
      Int elem = head(post);
      prev = appendright(prev, elem);
      post = tail(post);
      i = i + 1;
    }
    prev = appendright(prev, value);
    post = tail(post);
    this.list = concatenate(prev, post);
  }
}
```

La implementación del setter pasa por iterar sobre la lista hasta encontrar la posición solicitada para posteriormente concatenar dos listas con el elemento modificado en medio.

Una implementación similar se puede hacer para el otro tipo de nuestro lenguaje CABS: los booleanos.

El hecho de tener ahora nuestro tipo array encapsulado en una clase ABS nos permite utilizarlo con facilidad para implementar los arrays globales. Para este propósito podemos crear un método *init* que se encargue de la creación de los objetos array y unos métodos *getters* y *setters* modificados que se encarguen de realizar las llamadas a los métodos finales del array.

El código generado para el siguiente ejemplo

```
int array[10];
```

quedaría de este modo:

```
interface GLOBAL {
 Int getarray(Int indx);
 Unit setarray(Int indx, Int val);
 Unit init();
}

class GlobalVariables() implements GLOBAL {
  ArrayInt array;

  Unit init() {
    array = new ArrayIntC(10);
  }

  Int getarray(Int indx) {
    return await array!getV(indx);
  }

  Unit setarray(Int indx, Int val) {
    await array!setV(indx, val);
  }
}

...

{
  GLOBAL globalval = new GlobalVariables();
  await globalval!init();
  Intmain prog = new Impmain(globalval);
  await prog!main();
}
```

Por último, sería conveniente (y lo será más adelante en la traducción de las funciones) tener un método que nos devuelva el array global para usarlo con algunos fines locales. Esto se consigue con un método *retrieve* que implemente la clase $GlobalVariables$. Para el ejemplo anterior, quedaría como

```
interface GLOBAL {
 ArrayInt retrievearray();
 Int getarray(Int indx);
 Unit setarray(Int indx, Int val);
```

```
 Unit init();
}

class GlobalVariables() implements GLOBAL {
  ...
  ArrayInt retrievearray() {
    return array;
  }
  ...
}
...
```

2.2.1. Arrays multidimensionales

El lenguaje CABS permite en su sintaxis declarar arrays multidimensionales al estilo de C. ABS no cuenta con un tipo de datos similar, pero, gracias a la propuesta previamente expuesta, podemos simular el comportamiento de las matrices estableciendo una biyección con la representación de un array de tamaño el producto de las dimensiones de la matriz. En otras palabras, la traducción implementada en este trabajo considera las matrices como un array unidimensional.

A modo de explicación, sea *mat* una matriz multidimensional entera en $int^{N_1} \times \cdots \times int^{N_D}$ y supongamos que queremos acceder a la posición $(i_1, \ldots, i_D)$ donde $\forall j \in \{1, \ldots, D\}$ tenemos $i_j \in \{0, \ldots, N_j - 1\}$, entonces la posición p correspondiente a dicho elemento en un array en $int^{\prod_{i=1}^{D} N_i}$ vendría dada por $p = \sum_{i=1}^{D} i_i \cdot (\prod_{j=1}^{i} N_j)$ (cuadradar con implementación!!!!).

2.3. Expresiones aritméticas

CABS presenta una gran flexibilidad en sus expresiones aritmético-lógicas no presente en ABS. La sintaxis de ABS obliga a que el valor de retorno devuelto por una función solo pueda ser asignado a una variable, no pudiendo ser usado inmediatamente en una expresión del mismo tipo que el de retorno.

Esto nos obliga a traducir las expresiones aritméticas en una serie de asignaciones auxiliares que posteriormente se operan con los valores almacenados. Será por tanto necesario tener un modo de obtener nombres de variables auxiliares que no se referencien en ningún otro punto del programa traducido.

Sea por tanto $\mathbf{Aux} : \mathbb{N} \to \mathbf{Var}$ una función definidas sobre los enteros que devuelve el n-ésimo nombre de variable *libre*. En un sentido estricto esta función debería tomar como argumento el programa en CABS (y la traducción parcial) para saber qué nombres de variable son usados. A modo de simplificación podemos evitar esto reservando un conjunto de nombres de varibles para este propósito que no sean accesibles al usuario. Esta es la idea que posteriomente se llevará a cabo en la implementación del compilador.(Referencia!!!!)

De ahora en adelante nos referiremos a este conjunto como $\mathbf{Var}_R \subset \mathbf{Var}$.

Un posible ejemplo de subconjunto $\mathbf{Var}_R$ podría ser $\{aux\_var\_v : v \in \mathbb{N}\}$, que por tener la misma cardinalidad que $\mathbb{N}$ nos permite crear una biyección inmediata.

Teniendo estas herramientas, resulta fácil pensar en definir la traducción de las expresiones como una función que toma una expresión en CABS junto con un natural v y que devuelve una expresión en ABS junto con un natural que indique el siguiente natural no utilizado en la traducción. Si garantizamos no repetir un mismo n para dos expresiones distintas entonces garantizamos que las variables auxiliares no son usadas más allá de una única asignación y una única referencia.

Especificamos esta idea con la definición de las funciones $\mathcal{C}_{\mathbf{Aexp}} : \mathbf{Aexp} \to \mathbb{N} \to (\mathbf{ABS} \times \mathbb{N})$ y su homóloga $\mathcal{C}_{\mathbf{Bexp}}$ para las expresiones booleanas:

$$
\begin{aligned}
\mathcal{C}_{\mathbf{Aexp}}\,[\![n]\!]\,v &= (\mathit{Int}\,(\mathbf{Aux}\,v) = n, v+1)\\
\mathcal{C}_{\mathbf{Aexp}}\,[\![x]\!]\,v &= (\mathit{Int}\,(\mathbf{Aux}\,v) = x, v+1) \text{ donde } x \text{ es variable entera.}\\
\mathcal{C}_{\mathbf{Aexp}}\,[\![a_1 \odot_{\mathcal{A}} a_2]\!]\,v &= (c_1;\ c_2;\ \mathit{Int}\,(\mathbf{Aux}\,v'') = (\mathbf{Aux}\,(v'-1)) \odot_{\mathcal{A}} (\mathbf{Aux}\,(v''-1)), v''+1)\\
&\quad \text{donde } \mathcal{C}_{\mathbf{Aexp}}\,[\![a_1]\!]\,v = (c_1, v') \text{ y } \mathcal{C}_{\mathbf{Aexp}}\,[\![a_2]\!]\,v' = (c_2, v'')\\
\mathcal{C}_{\mathbf{Bexp}}\,[\![false]\!]\,v &= (\mathit{Bool}\,(\mathbf{Aux}\,v) = False, v+1)\\
\mathcal{C}_{\mathbf{Bexp}}\,[\![true]\!]\,v &= (\mathit{Bool}\,(\mathbf{Aux}\,v) = True, v+1)\\
\mathcal{C}_{\mathbf{Bexp}}\,[\![x]\!]\,v &= (\mathit{Bool}\,(\mathbf{Aux}\,v) = x, v+1) \text{ donde } x \text{ es variable booleana.}\\
\mathcal{C}_{\mathbf{Bexp}}\,[\![b_1 \odot_{\mathcal{B}} b_2]\!]\,v &= (c_1;\ c_2;\ \mathit{Bool}\,(\mathbf{Aux}\,v'') = (\mathbf{Aux}\,(v'-1)) \odot_{\mathcal{B}} (\mathbf{Aux}\,(v''-1)), v''+1)\\
&\quad \text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b_1]\!]\,v = (c_1, v') \text{ y } \mathcal{C}_{\mathbf{Bexp}}\,[\![b_2]\!]\,v' = (c_2, v'')\\
\mathcal{C}_{\mathbf{Bexp}}\,[\![a_1 \odot a_2]\!]\,v &= (c_1;\ c_2;\ \mathit{Bool}\,(\mathbf{Aux}\,v'') = (\mathbf{Aux}\,(v'-1)) \odot (\mathbf{Aux}\,(v''-1)), v''+1)\\
&\quad \text{donde } \mathcal{C}_{\mathbf{Aexp}}\,[\![a_1]\!]\,v = (c_1, v') \text{ y } \mathcal{C}_{\mathbf{Aexp}}\,[\![a_2]\!]\,v' = (c_2, v'') \text{ y } \odot \text{ comparador.}
\end{aligned}
$$