Capítulo 1

Introducción

La idea de programa ha cambiado mucho a lo largo de la historia de la informática. En un principio, la mayoría de sistemas solo podían hacerse cargo de una única secuencia de instrucciones a la vez, lo que durante muchos años fue una limitación para los programas concurrentes.

Hoy en día, con los avances en el hardware de las últimas décadas, las máquinas que usamos a diario permiten la ejecución simultanea de varios hilos de ejecución en sus procesadores multinúcleo. Es por esto que la mayoría de productos en el mercado están desarrollados con la idea de ejecutar varios hilos en paralelo.

El principal problema que presenta la ejecución en paralelo es la presencia de una memoria compartida sobre la que los distintos hilos de ejecución pueden realizar modificaciones en cualquier instante de tiempo. Asociada a esta situación se encuentran los principales problemas de la programación concurrente como son los deadlocks, las carreras de datos o comportamientos impredecibles del programa.

Esto lleva a que la cantidad de errores presentes en un programa concurrente sea mucho mayor que la que uno pueda cometer al desarrollar un programa con un solo hilo de ejecución. Sobre estos problemas se han abordado diferentes soluciones como pueden ser el uso de semáforos y cerrojos o implementaciones de algoritmos de más bajo nivel como el tie-breaker, pero el uso de estos mecanismos puede ser bastante complejo.

Es por esto, que en el terreno de la programación concurrente se sigue investigando para encontrar un modo definitivo que permita solucionar los problemas en el desarrollo de aplicaciones o que al menos permita indicar al programador si su código presenta posibles errores relacionados con una mala gestión de la concurrencia.

Teóricamente, un programa se considera concurrente cuando cuenta con varias secuencias de instrucciones que pueden actuar sobre una memoria o estado y en el que la selección de qué hilo o secuencia se ejecutará a continuación es escogida con cierta arbitrariedad. Más allá de tener varias instrucciones ejecutándose a la vez en el mismo procesador, nos podemos limitar a que solo una única instrucción se ejecuta a la vez, pero, desconocemos el orden en el que se ejecutará con respecto al resto de instrucciones. Esta noción es co-

nocida como *interleaving* o entrelazamiento.

Uno de los primeros asuntos estudiados es determinar, por ejemplo, el estado final al que se llega ejecutando un programa concurrente. Se entiende que la presencia de interleavings introduce cierta incertidumbre como podemos ver en el siguiente ejemplo:

```
int var;

proc f1() {
  var := 1;
}

proc f2() {
  var := 2;
}
```

Supongamos que $f1$ y $f2$ se ejecutan en paralelo. Cada una de ellas cuenta con una única instrucción de asignación sobre la variable *var*. Nuestro ordenador elegirá cuál de las dos secuencias ejecutar en el siguiente paso de un modo aleatorio. Digamos que primero se escoge la secuencia de $f1$ con lo que llegamos a un estado en el que $var = 1$. A continuación elegirá $f2$, puesto que la otra secuencia se habrá quedado vacía, y destruirá el valor anteriormente asignado llegando al estado en el que $var = 2$.

Sin embargo, si la decisión hubiera sido tomada al contrario y la primera secuencia ejecutada fuera la de $f2$, el valor destruido sería el 2 y terminaríamos con $var = 1$. ¿Qué estado produce nuestro programa entonces? La respuesta es que ambos, dependiendo de la ejecución. ¿Generalmente queremos esta situación? La respuesta suele ser que no.

En este ejemplo determinar los posibles estados finales de un programa es sencillo, pero no es el caso general. Los programas concurrentes padecen de una explosión exponencial en el número de interleavings en proporción al número de hilos y al número de instrucciones con los que cuentan.

¿Y esta explosión exponencial afecta también al número de estados posibles a lo que se puede llegar con la ejecución de un programa concurrente cualquiera? La respuesta es depende. Veamos otro ejemplo.

```
int var1;
int var2;

proc f1() {
  var1 := 1;  # 1
  var1 := 2;  # 2
}

proc f2() {
  var2 := 2;  # 3
  var2 := 1;  # 4
```

```
12  }
```

En esta ocasión podemos notar que cada uno de los procedimientos asigna únicamente a una de las variables luego el estado final al que llega este programa es al que cumple que $var1 = 2$ y $var2 = 1$. Sin embargo, el número de interleavings es 5, los que se corresponden con las secuencias "$1, 2, 3, 4$", "$1, 3, 2, 4$", "$3, 1, 2, 4$", "$3, 1, 4, 2$" y "$3, 4, 1, 2$".

Luego aparentemente no todos los interleavings tienen la misma importancia y algunos de ellos (o en el caso del ejemplo anterior todos) son equivalentes entre sí lo que resulta en que estudiar todas las posibles ejecuciones a veces puede no implicar una amplia variedad de estados finales.

Pero, por el momento, olvidémonos de cuándo o no es necesario analizarlas todas ellas y pensemos en cómo de difícil puede presentársenos este problema.

Empecemos por una instrucción como

```
1  int var = var1 + var2;
```

El significado que se puede dar de esta instrucción depende de qué consideramos que se puede ejecutar en paralelo y qué no. Es lo que se conoce como grano del paralelismo.

Generalmente se hablan de dos opciones de grano. El grano grueso, considera que la suma de las variables del ejemplo se computa y se asigna en único paso. Por otro lado, el paralelismo de grano fino considera que en un primer paso se lee el valor almacenado en $var1$, en un segundo paso se procede del mismo modo con $var2$, en un tercer paso se computaría la suma de ambos valores y, por último, en un cuarto paso se asignaría el resultado a var.

Claramente, el número de interleavings con grano fino será mayor que con una política de grano grueso. ¿Cuál de estas dos por tanto es la que se suele usar? Depende. Las implementaciones reales de los ordenadores actuales suelen considerar un paralelismo de grano fino. Por ejemplo, los procesadores que soportan un conjunto de instrucciones de ARM, solo permiten operar con valores guardados en registro y por tanto deben realizar cada uno de los pasos anteriores contando el cargar valores de memoria a registro o viceversa más la ejecución de una instrucción aritmética. Este es el caso también de algunos lenguajes de programación como C/C++ o Java.

También existe la posibilidad de controlar este comportamiento en un lenguaje de alto nivel en el que se garantice un grano grueso de paralelismo. De hecho, si aislamos la memoria de cada uno de los hilos de programa, el comportamiento de cualquiera de los dos granos es indiferente. Pero, ¿qué sentido tiene que cada uno de los hilos se ejecute con su propia región de memoria? Por si solo no tiene mucho sentido. Lo único que tendríamos son varios programas secuenciales que no interaccionan entre ellos y que a efectos teóricos no cuentan con ninguna concurrencia.

No obstante, ¿qué ocurre si somos capaces de hacer que cada hilo sea capaz de comunicarse con el resto de secuencias mandando valores por un canal? En ese caso, sí tenemos

algo más interesante. Este concepto es conocido como paso de mensajes.

Cada hilo de ejecución ahora podrá en un momento dado escuchar el canal o escribir en él y modificar su comportamiento según los mensajes que reciba. Seguimos por tanto contando con los mismos problemas de la concurrencia reduciendo eso sí el número de interleavings a tener en cuenta.

Imaginemos ahora que los mensajes mandados entre objetos son los que provocan la ejecución de los métodos que estos contienen. Esta es la idea en los modelos de concurrencia basados en actores.

En estos modelos, cada objeto ejecuta sus tareas de forma concurrente con respecto a las del resto de objetos con la única restricción de que cada objeto solo puede ejecutar una única tarea a la vez. El resto de tareas esperan en una cola cuyo orden en principio no es determinable. El paso de mensajes indica qué método desea ejecutar un objeto (pudiendo ser uno propio o perteneciente a otro objeto) y, dependiendo del tipo de llamada, una tarea puede dejar paso a otra si aún no cuenta con los valores necesarios para proseguir. Se trata de una concurrencia donde el scheduler o planificador no puede desasignar a una tarea dentro de un objeto si esta no ha terminado o si no ha llegado a un punto de espera, funcionando cada objeto como una especie de monitor. Esto es lo que se conoce como un modelo *non-preemptive*.

Al igual que como hemos comentado anteriormente, al introducir los modelos de paso de mensajes, el modelo de actores también reduce notablemente el número de interleavings a tener en cuenta. Esto es de vital importancia cuando se intentan aplicar técnicas de testing sistemático en las que se deben de tener en cuenta todos los entrelazamientos posibles y donde hay que tener muy en cuenta la exposión de estados, que hace que sean intratables en los casos generales. Para estos métodos de exploración sistemática se puede suponer que todas las instrucciones de una tarea se ejecutan una detrás de otra sin ningún entrelazamiento hasta que no se llega al final de la función en un *return*.

En este tipo de concurrencia se basan muchos lenguajes de programación como por ejemplo Erlang y Scala, además de ABS, el que va a ser nuestro compañero de viaje en este trabajo.

La motivación de este trabajo es la creación de un lenguaje de programación básico, llamado CABS, que permita recrear una concurrencia entre procesos a nivel de grano fino y sobre el que podamos emplear las potentes herramientas ya elaboradas para el lenguaje ABS en la materia del análisis de programas concurrentes. CABS será un lenguaje de programación con una sintaxis parecida a la de C con tipado estricto y estático, que incluye la posibilidad de usar funciones y arrays y que se mueve en el paradigma de la programación imperativa.

Sobre esta temática se han hecho ya trabajos similares de creación de lenguajes o compiladores como en [1]. En dicho trabajo, se abordo la detección de deadlocks en un lenguaje básico que contaba con primitivas para declarar procesos y cerrojos y que empleaba la

herramienta SACO desarrollada para ABS sobre una traducción formal. De un modo similar, en nuestro trabajo emplearemos las mismas técnicas formales sobre semánticas para demostrar que la traducción que se propondrá para CABS en efecto cumple con la preservación de todos los interleavings posibles y que, por tanto, los estados finales alcanzables ejecutando CABS con su semántica son los mismos que obtendríamos con ABS. A diferencia de [1], nuestro trabajo pretende llegar a desarrollar un lenguaje de programación Turing completo sobre el que se puedan implementar algoritmos y no restringirnos a crear una demostración meramente académica sobre la posibilidad de obtener los entrelazamientos en un lengueje de prueba.

De un modo similar al que se pretende hacer con CABS, en los años 80 se creo SPIN [1]. SPIN es una herramienta de verificación de sistemas distribuidos desarrollada por Gerard J. Holzmann que trabajaba sobre modelos escritos en Promela [2], un lenguaje de modelado, que era traducido a C. Era sobre el código en C donde se empleaban distintas técnicas de verificación, entre las que se incluía el uso de *Partial Order Reduction* (POR). En este aspecto, nosotros podremos aprovechar la heramiento SYCO, disponible para ABS y que implementa técnicas avanzadas de *Dynamic Partial Order Reduction* (DPOR). Sobre esta herramienta dedicaremos un capítulo completo al final de este trabajo. El objetivo de DPOR es conseguir reducir el número de estados de exploración necesarios para conocer los posibles resultados finales de un programa. Como hemos visto en los ejemplos anteriores, existen ocasiones en que el orden en la ejecución de dos instrucciones pertenecientes a procesos distintos no influye en el resultado final del programa y es en este aspecto donde DPOR consigue determinar que ordenes son redundantes para intentar salvar la explosión exponencial de estados intermedios.

En un ámbito más general podemos encontrar investigaciones como en [2], donde el uso de estas técnicas se emplean para el análisis de Redes definidas por software o SDN por sus siglas en inglés. SDN es un paradigma sobre la arquitectura de redes que permite un control sobre el comportamiento de las mismas. En [2] se establece una relación formal entre las SDN y el modelo de actores para la verificación de software distribuido, realizando una especificación en ABS.

Como hemos podido ver, el terreno de la verificación formal de programas concurrentes presenta ejemplos similares a los que intentaremos acometer en este trabajo que dividiremos en varios capítulos, cada uno de ellos centrado en un aspecto concreto. En los siguientes capítulos describiremos la sintaxis y semántica de nuestro lenguaje así como la del lenguaje ABS sobre el que realizaremos la traducción de CABS. Posteriormente se demostrará la idea de la corrección de la traducción, lo que nos permitirá extrapolar las propiedades del código ABS traducido al código original en CABS. Entre estas propiedades pueden encontrarse las resultantes del uso de herramientas formales desarrolladas sobre ABS. Tras toda la formalización, hablaremos de la implementación de la traducción propuesta en un compilador de CABS a ABS, que desarrollaremos en Java usando JLex y CUP, herramientas empleadas en la asignatura de Procesadores del Lenguaje. Por último, hablaremos de la aplicación principal del compilador de CABS que es el uso de la

[1] `https://en.wikipedia.org/wiki/SPIN_model_checker`
[2] `https://en.wikipedia.org/wiki/Promela`

herramienta SYCO para obtener todos los posibles resultados finales de la ejecución de un programa concurrente aprovechando el *state of the art* en DPOR.

¡Comencemos!

Capítulo 2

Sintaxis y semántica

2.1. Sintaxis de CABS

La mayoría de lenguajes de programación del mercado siguen una sintaxis común similiar a la que tiene el lenguaje original en el que se suelen basar que es C. A la hora de definir la sintaxis de CABS tomaremos como referencia de nuevo la de ese lenguaje.

Para sentar las bases de la notación que se usará de ahora en adelante para definir la semántica de nuestro lenguaje, diremos que **P** es un programa en CABS formado por instrucciones globales como lo son las declaraciones de variables globales y las declaraciones de funciones.

La declaración de variables constarán de un tipo y de un nombre de variable seguido de un punto y coma. Las funciones estarán formadas por un tipo de rotorno, un nombre de función, una lista de argumentos (es posible que sea vacia), un cuerpo con instrucciónes $S \in$ **Stm** y una expresión de retorno. A modo de ilustración podemos ver el siguiente código

```
type global_var1;
type global_var2;

type nombre_de_funcion(args...) {
  ...
  codigo
  ...
  return exp
}
```

donde se muestra la declaración de dos variables globales y de una función.

Los tipos que manejaremos en nuestro lenguaje serán enteros y booleanos, cuyos identificadores serán *int* y *bool* respectivamente. Permitiremos además la declaración y el uso de arrays de enteros y booleanos en asignaciones y expresiones aritmético-lógicas y como argumentos de funciones. No se podrán usar como tipo de retorno. Un ejemplo del uso de arrays es el siguiente:

```
int array[10]; # Array global con 10 enteros
int global_var; # Aventuramos que valdra 27, pero eso
   dependera de la semantica :)

int f(int res, int arr[10]) {
    int var;
    var = 2 * res + arr[1];
    return var;
}

int main() {
    array[0] = 10;
    array[1] = 5;
    int res;
    res = array[0] + 1;
    global_var = f(res, array);
    return 0;
}
```

Las instrucciones S del cuerpo de una función de un programa son las típicas de un lenguaje imperativo, entre las que se encuentran las asignaciones, las operaciones aritmetico lógicas, las instrucciones de control, como los condicionales y los bucles y, la clave de un lenguaje con concurrencia, un **thread** para la ejecución de funciones en paralelo imitando el comportamiento de la librería pthread en C. Veamos un ejemplo que use esta última construcción:

```
int var;

int main() {
    thread f(1);
    thread f(2);
    return 0;
}

void f(int value) {
    var = value;
}
```

Como podemos ver, las llamadas concurrentes son similares a las llamadas a procedimientos habituales precedidas por la palabra reservada **thread**.

2.2. Semántica de CABS

A continuación pasamos a hablar de la semántica del lenguaje CABS. La idea es dar un significado al código CABS de modo que quede definido el comportamiento de cada una de las construcciones presentes en el lenguaje. Para este propósito escogeremos una semántica de paso corto en la que la derivación se puede interpretar como una secuencia

de pasos que simulan las transiciones que generaría un código ejecutado en un ordenador real, es decir, los cambios en memoria y en la instrucción actual marcada por un contador de programa.

En una primera sección daremos las definiciones básicas de lo que serán los estados de nuestro programa para, posteriormente, especificar cuales serán las reglas que marcaran las transiciones entre ellos. Por último, mostraremos algunos ejemplos de derivación a partir de unos programas básicos.

2.2.1. Preámbulo semántico

Los programas en CABS se pueden entender, de forma simplificada, como unas secuencias de pasos que van a menejar unos valores enteros y lógicos posiblemente almacenados en una memoria donde también se guardarán los resultados desprendidos de las operaciones que se realicen. Es por esto que en primer lugar tenemos que definir un conjunto de valores $\mathbb{V} = \mathbb{Z} \cup \mathbb{B}$ que sea la unión de los enteros y de los booleanos, los elementos básicos de todo programa.

Una primera parte de la representación del estado estará formada por las variables globales de nuestro programa. Definimos $G = \mathbf{Var} \hookrightarrow \mathbb{V}$ 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. El conjunto **Var** contiene todos los nombre de variable posibles y usaremos $\mathbf{G}\,var$ para referirnos al valor almacenado por la variable global var. Por ser una función parcial quedan reflejadas en **G** únicamente las variables que han sido previamente declaradas.

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 $F = \mathbf{Func} \hookrightarrow (\mathbf{T} \times \mathbf{Stm} \times \mathbf{Args} \times (\mathbf{Exp} \cup \{\varepsilon\}))$ 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 $\text{arg} \in \mathbf{Args}$ 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. De nuevo tenemos un conjunto de nombres de funciones **Func**. En principio los nombres de variable y de función son los mismos en los lenguajes de programación habituales y es también el caso de CABS, no obstante, por simplicidad, consideraremos en la semántica que son conjuntos separados. A su vez, contamos también con la ventaja de las funciones parciales en F que solo tendrán la información de aquellas funciones que en verdad hayan sido definidas en el programa. **F** será la metavariable que usaremos para referirnos a una función de F concreta.

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 $\mathbf{State} = G \times F \times RP$ como una tupla que recoja la información de las variables globales y de las funciones

definidas en nuestro programa **P**, así como, una lista de marcos de ejecución o runtime processes que contendrá la información local de los procesos. Esto último quedará recogido en los elementos de $RP = ((\mathbf{Loc})^+ \times \mathbf{Stm})^*$ donde $\mathbf{Loc} = \mathbf{Var} \hookrightarrow \mathbb{V}$ será el conjunto de ámbitos locales. Por lo general usaremos la notación $local : s \in (\mathbf{Loc})^+$ para referirnos a la pila de llamadas de un proceso, donde $local \in \mathbf{Loc}$ y $s \in (\mathbf{Loc})^*$, emulando la notación de lista de un lenguaje funcional como Haskell. Usaremos además la metavariable **RP** para referirnos a una lista de marcos de ejecución concreta y el operador $\rightsquigarrow$ para referirnos a un elemento de la lista cualquiera, de modo que, $\mathbf{RP} \rightsquigarrow (local : s, S)$ indica que la lista de marcos **RP** contiene en concreto el marco $(local : s, S)$ donde S es el código del proceso y $(local : s)$ su pila de ámbitos locales.

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*, basándonos en la definición que hemos dado de estado.

La función init.

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

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

Con esta función conseguimos crear el ámbito global de variables y de funciones de nuestros programas en CABS.

La función (regla) start.

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

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

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).

Las primeras reglas que queremos definir serán las de las expresiones aritmético-lógicas. Nuestro lenguaje permite la llamada a funciones con valor de retorno, lo que hace que

tengamos que, desde un principio, manejar unas reglas que garanticen la no terminación de la evaluación de las expresiones. Será por ello que tengamos que dar una definición de una función parcial que dada una expresión nos devuelva otra expresión más simplificada, pudiendo emplear el estado del programa para ello y permitiendo modificaciones del mismo, hasta eventualmente quedarnos con un valor $v \in \mathbb{V}$, lo que consideramos la expresión más simplificada.

Empecemos por dar una definición en el caso de las expresiones enteras. 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:

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

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

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

donde x es variable entera.

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

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

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

donde $\bigodot$ es alguno de los operadores del lenguaje que tienen su homólogo semántico $\bigodot_{\mathcal{N}}$.

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

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

$$[\text{call}^{1}_{\mathcal{A}}] \ \frac{\mathbf{F}(\mathbf{func}) = (int, S_F, args_F, a) \qquad check\_args(args_F, args) \qquad eval(args) = args'}{\langle \mathbf{func}(args), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Aexp} \langle \mathbf{func}(args'), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle}$$

$$[\text{call}^2_{\mathcal{A}}]\ \frac{\mathbf{F}(\mathbf{func}) = (int, S_F, args_F, a) \quad check\_args(args_F, args) \quad eval(args) = v_1 : \cdots : v_n}{\langle \mathbf{func}(args), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Aexp} \langle \text{UNSTACK}\,(a)\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathbf{nil}\,[args \mapsto v_1 : \cdots : v_n] : local : s, S_F; S))\rangle}$$

donde *check_args* es una función auxiliar que comprueba la corrección de tipos y el número de argumentos y donde *eval* indica que los argumentos de la función son evaluados hasta conseguir los valores de $\mathbb{V}$ finales, haciendo uso de las reglas de la semántica de expresiones. De este modo se justifica que para ejecutar una función sea necesario evaluar uno a uno los argumentos de la función. De hecho, *eval* puede hacer uso de las reglas para expresiones booleanas en el caso de las funciones con parámetros mixtos. Dichas reglas son las que definen $\mathcal{B} : (\mathbf{Bexp} \times \mathbf{State}) \hookrightarrow (\mathbf{Bexp} \times \mathbf{State})$.

$$[\text{True}_{\mathcal{B}}]\ \frac{}{\langle true, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle \text{true}, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle}$$

$$[\text{False}_{\mathcal{B}}]\ \frac{}{\langle false, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle \text{false}, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle}$$

$$[\text{var}^L_{\mathcal{B}}]\ \frac{local(x) = v}{\langle x, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Bexp} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle}$$

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

donde x es una variable booleana.

$$[\textstyle\bigodot^1_{\mathcal{B}}]\ \frac{\langle b_1, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle b_1', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S'))\rangle}{\langle b_1 \bigodot b_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle b_1' \bigodot b_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S'))\rangle}$$

$$[\textstyle\bigodot^2_{\mathcal{B}}]\ \frac{\langle b_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Bexp} \langle b_2', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}{\langle v \bigodot b_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)))\rangle \rightarrow_{Bexp} \langle v \bigodot b_2', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')))\rangle}$$

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

donde $\bigodot$ es alguno de los operadores del lenguaje que tienen su homólogo semántico $\bigodot_{\mathcal{B}}$ booleano.

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

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

$$\left[\bigodot^3_{\mathcal{AB}}\right] \frac{}{\langle v_1 \bigodot v_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle v_1 \bigodot_{\mathcal{B}} v_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle}$$

donde $\bigodot$ es alguno de los operadores del lenguaje que tienen su homólogo semántico $\bigodot_{\mathcal{AB}}$ de comparación de enteros.

$$\left[\text{unstack}^1_{\mathcal{B}}\right] \frac{\langle b, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle b', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S'))\rangle}{\langle \textsc{unstack}\,(b)\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S))\rangle \rightarrow_{Bexp} \langle \textsc{unstack}\,(b')\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S'))\rangle}$$

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

$$\left[\text{call}^1_{\mathcal{B}}\right] \frac{\mathbf{F}(\mathbf{func}) = (bool, S_F, args_F, a) \quad check\_args(args_F, args) \quad eval(args) = args'}{\langle \mathbf{func}(args), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Bexp} \langle \mathbf{func}(args'), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle}$$

$$\left[\text{call}^2_{\mathcal{B}}\right] \frac{\mathbf{F}(\mathbf{func}) = (bool, S_F, args_F, a) \quad check\_args(args_F, args) \quad eval(args) = v_1 : \cdots : v_n}{\langle \mathbf{func}(args), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))\rangle \rightarrow_{Bexp} \langle \textsc{unstack}\,(a)\,, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathbf{nil}\,[args \mapsto v_1 : \cdots : v_n] : local : s, S_F; S))\rangle}$$

que repiten de forma similar lo que ya teníamos para la llamada de funciones ariméticas.

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

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

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

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

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

$$[\mathrm{ass}_{\mathrm{C}}^{3}] \frac{is\_global(var)}{(\mathbf{G},\mathbf{F},\mathbf{RP} \rightsquigarrow (local : s,\ \mathrm{var} = v; S)) \to (\mathbf{G}\,[\ \mathrm{var} \mapsto v]\,,\mathbf{F},\mathbf{RP} \rightsquigarrow (local : s, S))}$$

donde *var* es una variable entera. Para variables boolenas las regla es análoga.

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

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

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

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

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

$$\left[\text{thread}^1_{\text{C}}\right] \quad \frac{\mathbf{F}(\mathbf{func}) = (t, S_F, args_F, e) \qquad check\_args(args_F, args) \qquad eval(args) = args'}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, \mathbf{thread\ func}(args); S)) \rightarrow (\mathbf{G}, \mathbf{F}, (\mathbf{RP} \rightsquigarrow (s, \mathbf{thread\ func}(args'); S))}$$

$$\left[\text{thread}^2_{\text{C}}\right] \quad \frac{\mathbf{F}(\mathbf{func}) = (t, S_F, args_F, e) \qquad check\_args(args_F, args) \qquad eval(args) = v_1 : \cdots : v_n}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, \mathbf{thread\ func}(args); S)) \rightarrow (\mathbf{G}, \mathbf{F}, (\mathbf{RP} \cup (\mathbf{nil}\,[args \mapsto v_1 : \cdots : v_n] : [], S_F)) \rightsquigarrow (s, S))}$$

Capítulo 3

ABS: sintaxis y semántica

ABS (REFERENCIAS!!!) es un lenguaje de modelado para sistemas distribuidos con orientación a objetos y concurrencia basada en actores, donde los objetos de una clase ejecutan sus métodos en función de los mensajes recibidos por otros actores.

Sobre este lenguaje se han construido una serie de herramientas para el análisis de programas concurrentes que estamos interesados en mantener para nuestro lenguaje CABS, mediante una traducción de código.

Antes de poder continuar con el trabajo de traducción, es necesario hacer una introducción de las construcciones básicas de ABS.

3.1. Sintaxis

Como muestra su manual (ref!!!), ABS es un lenguaje que permite el uso de una amplia variedad de construcciones que van desde la definición de tipos de datos algebraicos hasta la definición de funciones dentro de los métodos de una clase.

De todas estas construcciones, para este trabajo emplearemos principalmente la definición de clases e interfaces del lenguaje.

La sintaxis para la declaración de interfaces en ABS es

```
interface nombre_de_interfaz {
  ...
  signaturas
  ...
}
```

donde las signaturas de los métodos son de la forma

```
Tipo nombre_metodo(args);
```

donde los argumentos son una lista separada por comas de cero o más nombres de variables precedidos de su tipo (al estilo de Java).

De los tipos predefinidos en ABS solo usaremos los enteros (*Int*) y los booleanos (*Bool*). También emplearemos el tipo genérico List para la construcción de Arrays.

La declaración de clases en ABS es similar a la de Java con la peculiaridad de que todas las clases definidas por el usuario deben implementar una interfaz. La sintaxis para la definición de clases en ABS es

```
class nombre_de_clase(args) implements nombre_de_interfaz {
  ...
  declaraciones de atributos privados
  ...
  ...
  implementaciones
  ...
}
```

La declaración de atributos de una clase es idéntica a la de Java, con la ausencia de las palabras reservadas *private*, *public* o *protected*. La visibilidad de todos los atributos es privada. Del mismo modo las implementaciones de métodos siguen el mismo estilo. Un método es por tanto público si está definido en la interfaz que implementa la clase, en caso contrario es privado.

Una característica especial de las clases de ABS es la posibilidad de definirlas con una lista de argumentos accesibles desde cualquier punto de la clase. Esta propiedad será ampliamente explotada con posterioridad para la implementación del concepto de variable global.

Por último nos queda discutir las llamadas a métodos de una clase. En ABS, las llamadas a métodos son un paso de mensaje, es decir, cuando un objeto llama a un método de otro objeto o de sí mismo, dicha llamada se mete en una cola a la espera de poder ser ejecutada por el objeto receptor. Un objeto (o actor) solo puede ejecutar un mensaje a la vez.

De los operadores de llamada a métodos de ABS, nosotros solo nos preocuparemos del operador '!', cuya sintaxis es

```
Fut<t_ret> ret = o!f(args);
```

La idea de esta llamada es mandar un mensaje al objeto o para que ejecute su método f. Esta llamada devuelve un tipo futuro. El tipo futuro permite entre otras cosas saber cuando se ha concluido la ejecución del mensaje y en este caso recuperar el valor de retorno del método.

Cuando queramos realizar una llamada síncrona, es decir, una llamada para la que no deseamos continuar la ejecución de un mensaje antes de conocer el valor de retorno del método llamado, emplearemos un *await*. La idea del await es paralizar la ejecución del mensaje actual, permitiendo a otros mensajes del objeto ser ejecutados, y esperar a que la variable futura de la llamada tenga un valor, es decir, que la llamada haya concluido. La construcción *await* tiene la siguiente sintaxis

```
await o!f(args);
```

y su tipo de retorno es el mismo que el del método de la llamada.

El resto de la sintaxis de ABS empleada en este trabajos se reduce al uso de los $if/else$ y del *while*, así como de la declaración de variables locales a los métodos y de asignaciones a varibles de los resultados de evaluación de expresiones aritmético-lógicas. La sintaxis referente a estas construcciones del lenguaje son prácticamente similares a las de Java.
El concepto de función *main* en ABS lo cumple un conjunto de instrucciones ABS escritas entre llaves y situadas al final del archivo del código.

3.2. Semántica

De forma similar al desarrollo expuesto para CABS, en esta sección seguiremos unos pasos similares para definir la semántica del lenguaje ABS.

Un estado de un programa en ABS vendrá representado por los objetos creados en ejecución y por la información estática aportada por las clases e interfaces, es decir, el código de la implementación de los métodos tanto públicos como privados y la inicialización de los atributos. Por tanto definimos $\mathbf{State_{ABS}} = \mathbf{O} \times \mathbf{C}$ donde los elementos de $\mathbf{O}$ serán una lista de objetos instanciados y los de $\mathbf{C}$ contendrán la definición de las clases e interfaces.

Un objeto vendrá dado a su vez por un identificador único o, su nombre de clase, una cola de mensajes a la que llamaremos **RT** (Runtime tasks), un identificador de tarea que indique que mensaje está procesando el objeto en ese instante y la información del estado de los atributos mediante $\mathbf{attr} : \mathbf{Var} \hookrightarrow \mathbb{V}_{\mathbf{ABS}}$, una función que asigne a un nombre de variable un valor de **ABS**. Puesto que los identificadores de objetos pueden venir dados por un número en $\mathbb{Z}$, se podría ver que $\mathbb{V}_{\mathbf{ABS}}$ es en esencia el mismo conjunto de valores que hemos definido para CABS según la implementación escogida.

El concepto de tarea recoge la información del ambito local $loc : \mathbf{Var} \hookrightarrow \mathbb{V}_{\mathbf{ABS}}$, el código del método $S \in \mathbf{Stm_{ABS}}$ y un identificador único de tarea.

Con estas herramientas básicas procedemos a dar una defínción formal de la semántica de ABS.

$$\frac{is\_local(x) \qquad is\_Int(x)}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, x = a; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[x \mapsto \mathcal{A}\left[\!\left[a\right]\!\right]_{loc,\mathbf{attr}}\right], S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{is\_attr(x) \qquad is\_Int(x)}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, x = a; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S, t), t, \mathbf{attr}\left[x \mapsto \mathcal{A}\left[\!\left[a\right]\!\right]_{loc,\mathbf{attr}}\right]), \mathbf{C})}$$

$$\frac{}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, Int\, x = a; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[x \mapsto \mathcal{A}\left[\!\left[a\right]\!\right]_{loc,\mathbf{attr}}\right], S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{is\_local(x) \qquad is\_Bool(x)}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, x = b; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[x \mapsto \mathcal{B}\left[\!\left[b\right]\!\right]_{loc,\mathbf{attr}}\right], S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{is\_attr(x) \qquad is\_Bool(x)}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, x = b; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S, t), t, \mathbf{attr}\left[x \mapsto \mathcal{B}\left[\!\left[b\right]\!\right]_{loc,\mathbf{attr}}\right]), \mathbf{C})}$$

$$\frac{}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, Bool\, x = b; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[x \mapsto \mathcal{B}\left[\!\left[b\right]\!\right]_{loc,\mathbf{attr}}\right], S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{\mathcal{B}\left[\!\left[b\right]\!\right]_{loc,\mathbf{attr}} = \text{TRUE}}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S_1 S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{\mathcal{B}\left[\!\left[b\right]\!\right]_{loc,\mathbf{attr}} = \text{FALSE}}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S_2 S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{while}(b)\{S_1\}S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{if}(b)\{S_1\mathbf{while}(b)\{S_1\}\}S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{\mathbf{C}\,[\![c']\!] = (\text{Inter}, attr_c, met, args_c) \qquad check\_args(args_c, args)}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \text{Inter } inter = new\, c'(args); S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} : o \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\,[inter \mapsto id'\,], S, t), t, \mathbf{attr}), \mathbf{C})}$$

donde $o = (id', c', [], \bot, \mathbf{attr'})$ con id' un nuevo identificador de objeto no utilizado y $\mathbf{attr'} = attr_c\left[args_c \mapsto \mathcal{E}\,[\![args]\!]_{loc,\mathbf{attr}}\right]$ los atributos del nuevo objeto creado.

$$\frac{loc \cup attr\,[\![inter]\!] = id' \qquad \mathbf{C}\,[\![c']\!] = (\text{Inter}, attr_c, met, args_c) \qquad contains(met, m) \qquad check\_args(args_m, args)}{(\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT'}, t', \mathbf{attr'})(id, c, \mathbf{RT} \rightsquigarrow (loc, inter!m(args); S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT'} : tsk, t', \mathbf{attr'})(id, c, \mathbf{RT} \rightsquigarrow (loc, S, t), t, \mathbf{attr}), \mathbf{C})}$$

donde $tsk = (loc', S', t'')$ con t'' un identificador de tarea nuevo y $loc' = \mathbf{nil}\,[args_m \mapsto args]$ y $met\,[\![m]\!] = (S', args_m)$

$$\frac{loc \cup attr\,[\![int]\!] = id' \qquad \mathbf{C}\,[\![c']\!] = (\text{Inter}, attr_c, met, args_c) \qquad contains(met, m) \qquad check\_args(args_m, args)}{(\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT'}, t', \mathbf{attr'})(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\,x = await\,int!m(args); S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow o'(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\,x = await\,t'; S, t), t, \mathbf{attr})}$$

donde $o' = (id', c', \mathbf{RT'} : tsk, t', \mathbf{attr'})$, $tsk = (loc', S', t'')$ con t'' un identificador de tarea nuevo y $loc' = \mathbf{nil}\,[args_m \mapsto args]$ y $met\,[\![m]\!] = (S', args_m)$

$$\frac{tsk = (loc', \varepsilon(\nu), t'')}{(\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT'} \rightsquigarrow tsk, t', \mathbf{attr'})(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\,x = await\,t'; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow o'(id, c, \mathbf{RT} \rightsquigarrow (loc\,[x \mapsto \nu\,], S, t), t, \mathbf{attr}), \mathbf{C})}$$

donde $o' = (id', c', \mathbf{RT'}, t', \mathbf{attr'})$

$$\frac{tsk = (loc', S, t'') \qquad S \neq \varepsilon(\nu)}{(\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT'} \rightsquigarrow tsk, t', \mathbf{attr'})(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\,x = await\,t'; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow o'(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\,x = await\,t'; S, t), \bot, \mathbf{attr}), \mathbf{C}}$$

donde $o' = (id', c', \mathbf{RT'}, t', \mathbf{attr'})$

$$\frac{}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S, t), \bot, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{return}\ a; S, t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \varepsilon(\mathcal{A}[\![a]\!]_{loc,\mathbf{attr}}), t), \bot, \mathbf{attr}), \mathbf{C})}$$

$$\frac{}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \varepsilon(\nu), t), t, \mathbf{attr}), \mathbf{C}) \to (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \varepsilon(\nu), t), \bot, \mathbf{attr}), \mathbf{C})}$$

3.3. Extensión semántica y sintáctica

Además de las definiciones anteriores asociadas al lenguaje ABS, puede resultar interesante añadir una serie de estructuras auxiliares a modo de azúcar sintáctico que puedan facilitarnos el camino en los siguientes capítulos de este trabajo.

Capítulo 4

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.

4.1. 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.

```
1 int var1;
2
3 void f() {
4     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 *suspend* tras llamar a f.

```
1   this!f();
2   suspend;
```

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 los cuerpos de f y *main* son un poco más largos como en el siguiente ejemplo?

```
1  int var1;
2  int var2;
3
4  void f() {
5      var1 = 2;
6      var2 = 4;
7  }
8
9  int main() {
10     thread f();
11     var1 = 1;
12     var2 = 3;
13     return 0;
14 }
```

En este caso no hay forma de regresar a la función *main* antes de que termine la ejecución de f suponiendo que se escoja esta tras el *suspend*. 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);
```

```
44    }
45  }
46
47  class Impmain(GLOBAL globalval) implements Intmain {
48    Int main() {
49      Intf aux_f = new Impf(globalval);
50      aux_f!f();
51      await globalval!setvar1(1);
52      await globalval!setvar2(3);
53      return 0;
54    }
55  }
56
57  {
58    GLOBAL globalval = new GlobalVariables();
59    Intmain prog = new Impmain(globalval);
60    prog.main();
61  }
```

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.

4.2. 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 encarge 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;
  }
  ...
}
...
```

4.2.1. 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!!!!).

4.3. Expresiones aritméticas (faltan arrays y funciones!!!!)

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:

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

Para las llamadas a funciones supondremos por el momento que las interfaces y clases asociadas se encuentran traducidas en algún otro punto del código y que por tanto podemos hacer uso de sus nombres.

$$\begin{aligned}
\mathcal{C}_{\mathbf{Aexp}}\,[\![f(e_1 \dots e_n)]\!]\,v^1 = (c_1 \dots c_n \\
&\mathrm{Int}f\,(\mathbf{Aux}\,(v^{n+1})) = \text{new Imp}f(\text{globalval}) \\
&Int\,(\mathbf{Aux}\,(v^{n+1}+1)) = (\mathbf{Aux}\,(v^{n+1}))!((\mathbf{Aux}\,(v^2-1)),\dots, \\
&(\mathbf{Aux}\,(v^{n+1}-1))), v^{n+1}+2) \\
&\text{donde } \mathcal{C}_{\mathbf{Exp}}\,[\![e_i]\!]\,v^i = (c_i, v^{i+1}) \text{ (escogiendo } \mathcal{C}_{\mathbf{Aexp}} \text{ o } \mathcal{C}_{\mathbf{Bexp}} \text{ según convenga)} \\
\mathcal{C}_{\mathbf{Bexp}}\,[\![f(e_1 \dots e_n)]\!]\,v^1 = (c_1 \dots c_n \\
&\mathrm{Int}f\,(\mathbf{Aux}\,(v^{n+1})) = \text{new Imp}f(\text{globalval}) \\
&Bool\,(\mathbf{Aux}\,(v^{n+1}+1)) = (\mathbf{Aux}\,(v^{n+1}))!((\mathbf{Aux}\,(v^2-1)),\dots, \\
&(\mathbf{Aux}\,(v^{n+1}-1))), v^{n+1}+2) \\
&\text{donde } \mathcal{C}_{\mathbf{Exp}}\,[\![e_i]\!]\,v^i = (c_i, v^{i+1}) \text{ (escogiendo } \mathcal{C}_{\mathbf{Aexp}} \text{ o } \mathcal{C}_{\mathbf{Bexp}} \text{ según convenga)}
\end{aligned}$$

4.4. Traducción de código

Tras esta introducción, podemos vislumbrar que aspecto tendrá la traducción de código llevada a cabo en este trabajo. Obviaremos el preámbulo de los programas ABS, que incluirá la definición de los clases Array, y nos centraremos en otros aspectos más importantes.

Sea pues la función $\mathcal{C} : \mathbf{CABS} \to \mathbb{N} \to (ABS \times \mathbb{N})$ nuestra función de traducción de CABS a ABS definida recursivamente del siguiente modo:

$$\begin{aligned}
\mathcal{C}\,[\![int\,var;]\!]\,v &= (Int\,var, v) \\
\mathcal{C}\,[\![bool\,var;]\!]\,v &= (Bool\,var, v) \\
\mathcal{C}\,[\![var = a;]\!]\,v &= (c; var = (\mathbf{Aux}\,(v'-1)), v') \\
&\quad \text{donde } \mathcal{C}_{\mathbf{Aexp}}\,[\![a]\!]\,v = (c, v') \text{ y } var \text{ variable local entera} \\
\mathcal{C}\,[\![var = b;]\!]\,v &= (c; var = (\mathbf{Aux}\,(v'-1)), v') \\
&\quad \text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v = (c, v') \text{ y } var \text{ variable local booleana} \\
\mathcal{C}\,[\![S_1 S_2]\!]\,v &= (c_1 c_2, v'') \text{ donde } \mathcal{C}\,[\![S_1]\!]\,v = (c_1, v') \text{ y } \mathcal{C}\,[\![S_2]\!]\,v' = (c_2, v'') \\
\mathcal{C}\,[\![\mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}]\!]\,v &= (c_1; \mathbf{if}(\mathbf{Aux}\,(v'-1))\{c_2\}\mathbf{else}\{c_3\}, v''') \\
&\quad \text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v = (c_1, v'),\ \mathcal{C}\,[\![S_1]\!]\,v' = (c_2, v'') \text{ y } \mathcal{C}\,[\![S_2]\!]\,v'' = (c_3, v''') \\
\mathcal{C}\,[\![\mathbf{while}(b)\{S\}]\!]\,v &= (c_1; \mathbf{while}(\mathbf{Aux}\,(v'-1))\{c_2 c_3\,\mathbf{Aux}\,(v'-1) = \mathbf{Aux}\,(v'''-1);\}, v''') \\
&\quad \text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v = (c_1, v'),\ \mathcal{C}\,[\![S]\!]\,v' = (c_2, v'') \text{ y } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v'' = (c_3, v''')
\end{aligned}$$

Sobre la traducción del while cabe destacar que es necesario traducir dos veces su condición y el uso de una asignación adicional. Esto es debido a que el concepto de while obliga a evaluar su condición antes de la ejecución de su cuerpo en cada una de las iteraciones para decidir si el salto se toma o no. Por el modo en que se han traducido las expresiones aritmético-lógicas, la condición en ABS de un while se reduce a comprobar el valor asignado a una variable auxiliar y es por esto por lo que, sobre dicha variable, se hará una

asignación adicional al final de toda iteración.

En segundo lugar, hace falta mencionar que las funciones de traducción de expresiones booleanas no contemplan la reutilización de variables, por lo que aunque la condición sea la misma, se emplearan unos nombres de variables nuevos al final del cuerpo del while respecto a los que se usaban al evaluar la condición fuera del cuerpo. A efectos prácticos, es fácil demostrar que los nombres de las variables auxiliares no influyen en el cómputo de la expresión booleana. De hecho se ve claramente que la traducción es la misma salvo un offset en el natural que identifica a las variables auxiliares.

Es por esto por lo que podemos entender que en la traducción del código se puede indistintamente sustituir el código c_3 por c_2 en la regla del while si por motivos de simplicidad hiciera falta a la hora de desarrollar la corrección de esta traducción. En una implementación real esto no es inmediato porque los compiladores o interpretes de ABS no permiten la declaración duplicada de variables, problema que en este trabajo no nos atañe a nivel abstracto.

Para la traducción de las funciones necesitamos previamente una traducción de los argumentos $\mathcal{C}_{arg} : \mathbf{Args} \to \mathbf{Args_{ABS}}$ que en esencia lo único que hace es cambiar los tipos de CABS a los de ABS. (Arrays!!!!)

$$\begin{aligned}\mathcal{C}_{arg}\,\varepsilon &= \varepsilon\\ \mathcal{C}_{arg}\,(int\,var,\,arg) &= Int\,var,\,\mathcal{C}_{arg}\,arg\\ \mathcal{C}_{arg}\,(bool\,var,\,arg) &= Bool\,var,\,\mathcal{C}_{arg}\,arg\end{aligned}$$

Usando esta traducción de argumentos tenemos que

$$\begin{aligned}\mathcal{C}\,[\![int\,\mathbf{func}(arg)\{S\,\mathbf{return}\ a\}]\!]\,v = (&\text{interface Int}\mathbf{func}\{Int\,\mathbf{func}(\mathcal{C}_{arg}\,arg);\}\\ &\text{class Imp}\mathbf{func}(\text{GLOBAL globalval}) \text{ implements Int}\mathbf{func}\{\\ &Int\,func(\mathcal{C}_{arg}\,arg)\{c\,c'\,\mathbf{return}\ \mathbf{Aux}\,(v''-1)\}\},v'')\\ &\text{donde } \mathcal{C}\,[\![S]\!]\,v=(c,v') \text{ y } \mathcal{C}_{\mathbf{Aexp}}\,[\![a]\!]\,v'=(c',v'')\\ \mathcal{C}\,[\![bool\,\mathbf{func}(arg)\{S\,\mathbf{return}\ b\}]\!]\,v = (&\text{interface Int}\mathbf{func}\{Bool\,\mathbf{func}(\mathcal{C}_{arg}\,arg);\}\\ &\text{class Imp}\mathbf{func}(\text{GLOBAL globalval}) \text{ implements Int}\mathbf{func}\{\\ &Bool\,func(\mathcal{C}_{arg}\,arg)\{c\,c'\,\mathbf{return}\ \mathbf{Aux}\,(v''-1)\}\},v'')\\ &\text{donde } \mathcal{C}\,[\![S]\!]\,v=(c,v') \text{ y } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v'=(c',v'')\\ \mathcal{C}\,[\![void\,\mathbf{func}(arg)\{S\}]\!]\,v = (&\text{interface Int}\mathbf{func}\{Unit\,\mathbf{func}(\mathcal{C}_{arg}\,arg);\}\\ &\text{class Imp}\mathbf{func}(\text{GLOBAL globalval}) \text{ implements Int}\mathbf{func}\{\\ &Unit\,func(\mathcal{C}_{arg}\,arg)\{c\}\},v') \text{ donde } \mathcal{C}\,[\![S]\!]\,v=(c,v')\end{aligned}$$

y para las variables globales tenemos

$$\begin{aligned}\mathcal{C}\,[\![var=a;]\!]\,v &= (c; await\ \text{globalval!set}var(\mathbf{Aux}\,(v'-1)),v')\\ &\quad\text{donde } \mathcal{C}_{\mathbf{Aexp}}\,[\![a]\!]\,v=(c,v') \text{ y } var \text{ variable global entera}\\ \mathcal{C}\,[\![var=b;]\!]\,v &= (c; await\ \text{globalval!set}var(\mathbf{Aux}\,(v'-1)),v')\\ &\quad\text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v=(c,v') \text{ y } var \text{ variable global booleana}\end{aligned}$$

Por último, hablaremos de la traducción del **thread**. Como ya hemos comentado en las secciones anteriores, la traducción propuesta es similar a la de las llamadas a función con la diferencia de que no se espera a la resolución del valor futuro de retorno. Su regla es

$$\begin{aligned}\mathcal{C}\,[\![\mathbf{thread}\, f(e_1 \dots e_n)]\!]\, v^1 = (& c_1 \dots c_n \\ & \mathrm{Int} f\ (\mathbf{Aux}\,(v^{n+1})) = \text{new Imp}f(\text{globalval}) \\ & (\mathbf{Aux}\,(v^{n+1}))!((\mathbf{Aux}\,(v^2-1)), \dots, (\mathbf{Aux}\,(v^{n+1}-1))), v^{n+1}+1) \\ & \text{donde } \mathcal{C}_{\mathbf{Exp}}\,[\![e_i]\!]\, v^i = (c_i, v^{i+1}) \text{ (escogiendo } \mathcal{C}_{\mathbf{Aexp}} \text{ o } \mathcal{C}_{\mathbf{Bexp}} \text{ según convenga)}\end{aligned}$$

Capítulo 5

Corrección

A lo largo de este capítulo usaremos las definiciones de las semánticas previamente expuestas para comprobar que la traducción propuesta de CABS a ABS es correcta. En otras palabras, durante las próximas secciones, demostraremos que dado un código en CABS y partiendo de un estado inicial podemos "ejecutar" una serie de pasos hasta llegar a un nuevo estado que tiene un homólogo en ABS y que resulta de ejecutar la traducción desde un estado equivalente al inicial en CABS (y viceversa).

Este procedimiento es conocido como bisimulación y nos permitirá hacer afirmaciones tan fuertes como las que buscamos, es decir, que si tenemos un código en CABS y dada su traducción sabemos, usando la herramientas ya desarrolladas para ABS, que cumple una cierta propiedad partiendo de un estado, sabremos entonces que el mismo resultado se sostiene para el código en CABS.

5.1. Equivalencia entre estados

Sea $(\mathbf{G}, \mathbf{F}, \mathbf{RP})$ un estado en CABS y $(\mathbf{O}, \mathbf{C})$ un estado en ABS. Diremos que estos estados son equivalentes si:

- dado $(id, c, \mathbf{RT}, t, \mathbf{attr}) \in \mathbf{O}$ el objeto correspondiente a la clase GlobalVariables, tenemos que las funciones $\mathbf{G}$ y $\mathbf{attr}$ son la misma.
- para todo nombre de función $\mathbf{func}$ con $\mathbf{F}(\mathbf{func}) = (t, S, args, a)$, tenemos que $\mathbf{C}[\![\text{Imp}\mathbf{func}]\!] = (\text{Int}\mathbf{func}, \mathbf{nil}, met, \text{arg}_c)$ donde $met\,[\![\mathbf{func}]\!]$ contiene la traducción de los argumentos $args$ y del código de la función y los argumentos de clase arg_c contienen solo una variable con el tipo de la interfaz de las variables globales.
- cada elemento en $\mathbf{RP}$ se corresponde con un conjunto de tareas en los objetos de $\mathbf{O}$, en concreto, cada entorno de variables apilado se corresponde con los atributos (obviando variables auxiliares) de una tarea cuyo código traducido se corresponde con un segmento de instrucciones del código del proceso. Lo que se quiere decir con esto es que, mientras que las llamadas a función en CABS se corresponden con el apilamiento de un nuevo entorno de variables y la concatenación del código de la función con el código actual del proceso, en ABS se crea una nueva tarea a la que se espera para obtener el valor de retorno.

- en caso de existir un elemento $(local : s, x = e; S) \in \mathbf{RP}$ donde e tiene elementos en $\mathbb{V}$ ya calculados, se tiene que en la tarea del estado equivalente se han ejecutado ya las asignaciones a las variables auxiliares correspondientes con los valores ya calculados y, por tanto, quedan por calcular las asignaciones a variables auxiliares restantes en la expresión. En otras palabras, cada paso de evaluación de una expresión será una asignación en la traducción en ABS. Esto es válido tanto para enteros como para booleanos.

5.2. Corrección de las expresiones aritmético-lógicas

Una de las peculiaridades que tiene CABS, es permitir en sus expresiones una notación para realizar llamadas a función con retorno de valor. Como ya notamos en el capítulo anterior, esto hace que la traducción se apoye en el uso de unas variables auxiliares que permiten dividir el cómputo de una expresión en el cómputo de las distintas subexpresiones que la componen.

Cada expresión utilizada en un programa CABS se corresponde de forma unívoca con una variable auxiliar en ABS que almacenará el valor de la expresión cuando este esté disponible. De este modo, en todo momento podemos hacer uso de esta información, aunque, por simplificar la notación, no esté presente en el estado del programa.

Hecha esta introducción, procedemos a comprobar en primer lugar que las asignaciones hechas en CABS se corresponden con las de la traducción de estas a ABS, llegando de estados equivalentes a estados equivalentes en un solo paso.

5.2.1. Variables locales

Caso $x = \nu$ **con** $\nu \in \mathbb{V}$

Sean $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = \nu; S))$ y $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ estados equivalentes, donde la variable auxiliar k-esima es la variable asignada a la expresión ν de forma que $loc(aux\_var\_k) = \nu$.

Tenemos que del estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = \nu; S))$, aplicando la regla $[\mathrm{ass}^2_{\mathrm{C}}]$, llegamos al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local\,[x \mapsto \nu] : s, S))$ en CABS. Por otro lado, del estado equivalente en ABS $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, x = aux\_var\_k; S, t), t, \mathbf{attr}), \mathbf{C})$, aplicando la regla de asignación local correspondiente, llegamos al estado $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[x \mapsto \mathcal{A}\left[\!\left[aux\_var\_k\right]\!\right]_{loc,\mathbf{attr}}\right], S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ y del hecho de que dicha variable auxiliar tenga el valor de ν llegamos inmediatamente a un estado equivalente.

Caso $x = n$

Sea $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = n; S))$ y sea $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, Int\, aux\_var\_k = n; x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ su estado equivalente, donde la variable auxiliar k-esima es la variable asignada a la expresión n.

En este caso, solo podemos aplicar para el proceso actual la regla $[\text{ass}^1_{\text{C}}]$ que delega la acción en las reglas para expresiones aritmético-lógicas. La regla aplicable en esta situación es $[\text{num}_{\mathcal{A}}]$ con lo que llegamos al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = \mathcal{N}[\![n]\!]; S))$.

En el lado de ABS aplicamos la regla de asignación sobre la variable auxiliar llegando al estado $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[aux\_var\_k \mapsto \mathcal{A}[\![n]\!]_{loc,\mathbf{attr}}\right], x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ preservandose la equivalencia entre estados al asignar el valor de la expresión a su variable auxiliar.

Caso $x = f(\cdots)$ (simplificación sin argumentos!!!)

Sean los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = f(); S))$ y $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \text{Intf } aux\_var\_(k-1) = new\,\text{Impf}(globalval); Int\, aux\_var\_k = await\, auxi\_var\_(k-1)!f(); x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ donde la variable auxiliar k-esima es la variable asignada a la expresión $f()$.

La semántica de CABS nos permite usar la regla $[\text{ass}^1_{\text{C}}]$ con la regla $\left[\text{call}^2_{\mathcal{A}}\right]$ resultando en el estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathbf{nil} : local : s, S_F\, x = \text{UNSTACK}\,(a)\,; S))$ donde a es la expresión de retorno de f.

Por otro lado, en la semántica de ABS, podemos aplicar primero la regla de creacción de objetos y a continuación la regla de llamada síncrona, resultando el estado

$$(\mathbf{O} \rightsquigarrow (id', c', (\mathbf{nil}, S^F_{\mathbf{ABS}}, t'), \bot, \mathbf{attr}') \\ (id, c, \mathbf{RT} \rightsquigarrow (loc, Int\, aux\_var\_k = await\, t'; x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

Llegados a este punto tenemos un estado que es equivalente al resultante en la semántica de CABS puesto que el concepto de apilar un nuevo ámbito de variables en ABS se ve reflejado creando un objeto nuevo que tiene como única tarea asignada el cómputo del código traducido de la función f. El objeto original se ve pausado por el *await* hasta que no termine de ejecutarse el código de f, imitando el concepto del código concatenado en CABS que no deja ejecutar resto de instrucciones hasta no terminar con las de la función.

Otro detalle a tener en cuenta es el indicador de tarea actual. Tras dar los dos paso anteriores en la semántica de ABS se nos lleva a tener como tarea asignada en el nuevo objeto a $\bot$. Evidentemente, antes de poderse ejecutar la primera instrucción de f es necesario que se ejecute la regla de selección de tarea. El hecho de que ningún objeto de nuestra traducción vaya a ejecutar más de una tarea (a excepción del objeto de variables globales del que hablaremos más adelante) nos permite extender la idea de equivalencia a este estado actual, puesto que consideramos que aplicar una regla de selección no afecta en esencia a la configuración actual en ABS ya que solo lo hacen los "movimientos" en el código.

De forma similar, el hecho de tener que dar dos pasos tampoco afecta, pese a existir una concurrencia en la semántica. El motivo es que entre ambos pasos el objeto creado solo es referenciado por una variable auxiliar de modo que se garantiza que no puede

ser modificado desde ningún otro punto de la ejecución. De este modo podemos volver a extender el concepto de equivalencia permitiendo que el paso intermedio de creación del objeto (omitido en este texto) pueda ser también considerado un estado que mantiene la equivalencia con $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathbf{nil} : local : s, S_F\, x = \textsc{unstack}\,(a)\,; S))$. Con esto se quiere decir que, al igual que las reglas de selección de tarea, la creación de objetos no altera tampoco el estado intrínseco del programa.

Caso $x = \textsc{unstack}\,(\nu)$ **con** $\nu \in \mathbb{V}$

Sean los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local' : local : s, x = \textsc{unstack}\,(\nu)\,; S))$ y

$$(\mathbf{O} \rightsquigarrow (id', c', (loc', \mathbf{return}\ aux\_var\_k', t'), t', \mathbf{attr}')$$
$$(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\, aux\_var\_k = await\, t'; x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

donde la variable auxiliar k'-esima es la variable asignada a la expresión ν, que tiene su valor asignado en loc', y la k-esima al unstack.

En el caso de CABS, aplicar la regla $[\text{ass}^1_{\text{C}}]$ con la regla $\left[\text{unstack}^2_{\mathcal{A}}\right]$ para la expresión nos lleva al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = \nu; S))$.

En el caso de ABS, podemos aplicar la regla de retorno que nos lleva a un estado intermedio

$$(\mathbf{O} \rightsquigarrow (id', c', (loc', \varepsilon(\mathcal{A}\,[\![aux\_var\_k']\!]_{loc,\mathbf{attr}}), t'), t', \mathbf{attr}')$$
$$(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\, aux\_var\_k = await\, t'; x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

Desde este estado se puede aplicar la regla del *await* que permite recoger el valor de ν y asignarselo, en este caso, a la variable aux_var_k, llegando al estado

$$(\mathbf{O} \rightsquigarrow (id', c', [], \bot, \mathbf{attr}')$$
$$(id, c, \mathbf{RT} \rightsquigarrow (loc\,[aux\_var\_k \mapsto \nu], x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

que mantiene la equivalencia.

De nuevo se nos presenta la discusión de los pasos múltiples. El mismo argumento que hemos presentado anteriormente es aplicable a esta situación.

Cabe destacar que el concepto de desapilar un entorno en CABS queda reflejado en el hecho de que el objeto con el ámbito superior no vuelve a ser usado y queda vacío de tareas.

Caso $x = a_1 \odot a_2$

Antes de discutir este caso conviene ver que en CABS el cómputo de $a_1 \odot a_2$ precisa del cómputo de a_1, de modo que el argumento que vamos a seguir para el caso compuesto es asumir que todo va bien si nos encontraramos con la expresión a_1 y que por tanto la asunción se puede emplear para la expresión $a_1 \odot a_2$.

Otro detalle importante a tener en cuenta es que no se empieza a procesar la expresión a_2 hasta que no hemos completado el cómputo de a_1, como se refleja en las reglas de la semántica de CABS. Esto se reflejará también en ABS ya que las asignaciones de la expresión a_2 vienen precedidas por las de la expresión a_1.

Dicho esto, sean los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = a_1 \odot a_2; S))$ y

$$(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, c_1 c_2\, Int\, aux\_var\_k \ = aux\_var\_k_1 \odot aux\_var\_k_2; \\ x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

donde c_1 es el código asociado a a_1, c_2 el asociado a a_2 y $aux\_var\_k_1$, $aux\_var\_k_2$ y aux_var_k las variables asociadas a cada una de las expresiones aritméticas en juego.

En concreto tenemos que la última instrucción de c_1, por el modo en que hemos definido la traducción, se corresponde con una asignación a su variable auxiliar. Por el modo en que están definidas las expresiones aritméticas, podemos aplicar inducción estructural con lo que, por hipotesis de inducción tenemos que los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = a_1; S))$ y $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, c_1\, x = aux\_var\_k_1; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ llegan en un solo paso a los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = a_1'; S))$ y $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, c_1'\, x = aux\_var\_k_1; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$.

Teniendo en cuenta esto y el lema (escribir lema!!! La idea es que si tienes como código S_1 y un estado s y se ejecuta un paso para llegar a S_1' y s' tenemos que de $S_1 S_2$ y s pasamos a $S_1' S_2$ y s'), llegamos a que de los estados originales llegamos en un paso a los estados $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = a_1' \odot a_2; S))$ y

$$(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, c_1' c_2\, Int\, aux\_var\_k \ = aux\_var\_k_1 \odot aux\_var\_k_2; \\ x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

que son equivalentes.

Casos $x = \nu \odot a_2$ y $x = \nu \odot \mu$ con $\nu, \mu \in \mathbb{Z}$

La situación del primer caso es completamente análoga a la del caso anterior con la diferencia de que la hipótesis de inducción es aplicada sobre la expresión a_2.

Para el segundo caso tenemos los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = \nu \odot \mu; S))$ y $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, Int\, aux\_var\_k = aux\_var\_k_1 \odot aux\_var\_k_2; x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ que de forma análoga al caso $x = n$ llegan en un paso a $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, x = \nu \odot_{\mathcal{N}} \mu; S))$ y

$$(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc \left[aux\_var\_k \mapsto \mathcal{A}\left[\!\left[aux\_var\_k_1 \odot aux\_var\_k_2\right]\!\right]_{loc,\mathbf{attr}}\right], \\ x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$$

que son equivalentes.

Casos $x = f(a_1, \ldots, a_n)$ y $x = \textsc{unstack}\,(a)$

Podemos considerar la llamada a una función con argumentos o la instrucción unstack como operadores aritméticos (n-ario el primero, unario el segundo) que del mismo modo que los anteriores operadores binarios esperan al computo de las subexpresiones para poder ejecutarse.

El proceder con estos es identico en el sentido de aplicar la hipótesis de inducción con las subexpresiones y componer de nuevo con el operador. La llamada final a la función cuando están preparados todos los operandos o la ejecución del unstack están reflejados ya en los casos base presentados al principio de la subsección.

5.2.2. Variables globales (TODO!!!)

5.3. Corrección del resto de instrucciones

Declaraciones de variable

Este caso es inmediato puesto que por definición en CABS una declaración inicializa la variable a cero (o false para los booleanos) a nivel semántico y en nuestra traducción forzamos esta situación con una declaración y una asignación. En ambas situaciones, con las reglas $\left[\text{Decl}_{\text{C}}^{int}\right]$ o $\left[\text{Decl}_{\text{C}}^{bool}\right]$ de CABS y las reglas (!!!) de ABS, llegamos en un paso a estados equivalentes.

Caso if

El if vuelve a presentar una situación similar a la que hemos tenido con las expresiones aritméticas y la que hemos omitido para las expresiones booleanas.

Tenemos que las dos reglas axiomáticas del if en CABS (la regla para el valor true y el valor false de la condición) son los casos base de un desarrollo inductivo como el que hemos comentado anteriormente y se corresponden con un único paso de ejecución en ABS con la regla del if. La tercera regla de CABS es sobre la que se emplea la hipótesis de inducción y se corresponde con los pasos de ejecución de la expresión booleana.

De este modo, es fácil ver que volvemos a tener que la ejecución de un paso en el if en CABS (ya sea en la condición o en la decisión de que rama se toma) se corresponde con un paso de ejecución (obviando pasos no relevantes, como los de cambio de tarea) en ABS.

Caso while

El problema que se nos presenta con el while es que su traducción a ABS no preserva la equivalencia de estados inmediatamente al aplicar las reglas de la semántica. De hecho, en CABS el while junto con su condición mutan en un if equivalente resultante de desenrollar el blucle una vez, mientras que, en ABS, la traducción empieza por computar las expresiones boolenas de la condición y cuando tiene el resultado listo se convierte en un

if para entonces sí volver a ser un estado equivalente.

Hay varias formas de solucionar este problema. Una de ellas pasaría por hacer una modificación en la semántica de CABS, creando una prórroga antes de realizar la conversión al if. Esta opción, aunque complicaría un poco más la semántica del lenguaje, nos permitiría tener de nuevo una equivalencia paso a paso como la que hemos tenido hasta ahora. Las nuevas reglas del while en CABS serían las siguientes:

$$\left[\text{while}_{\text{C}}^{\text{ALT}}\right] \frac{\langle b, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \varepsilon))\rangle \rightarrow_{Bexp} \langle b', (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathbb{B}}))\rangle}{\begin{array}{r}(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{while}(b)\{S_1\}S)) \rightarrow \\ (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathbb{B}}\, \mathbf{while}(b')\{S_1\}S))\end{array}}$$

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

donde ν es un valor booleano y b es la expresión booleana original de la condición del while.

Una de las primeras asunciones que se tienen en cuenta en estas reglas es la estatismo del código. A la hora de representar las reglas semánticas se ha optado por una semántica de paso corto en la que el código evoluciona indicando lo que queda de ejecución. Este no es el caso real de las implementaciones habituales en las que el código se mantiene inmutable. No obstante, y por motivos de simplicidad, generalmente, se opta por esta aproximación en la que el código muta. Es por ello que en la regla $[\text{while}_{\text{C}}^{\nu}]$ tenemos que asumir que podemos recuperar la condición booleana original para poder reescribir el bucle original en el interior del if. Asunción que por otra parte que no es muy exigente.

La segunda cuestión que nos debemos plantear es si sustituir la regla del while original por estas dos nuevas reglas no modifica el significado de los programas. La forma de ver que esto no ocurre pasa por usar inducción sobre la longitud de la secuencia de ejecución de la evaluación de la condición desde el estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{while}(b)\{S_1\}S))$, en la que solo se emplean las reglas del while y del if.

La idea es que si $b \in \mathbb{B}$ tendríamos en un solo paso la misma transición que llega al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{if}(\nu)\{S_1 \mathbf{while}(b)\{S_1\}\}; S))$, luego el caso base estaría resuelto. Para cadenas de longitud k suponemos que se cumple, tras la ejecución de esos k pasos en ambas semánticas, la igualdad de los estados finales y veamos que se sigue para cadenas de $k+1$ pasos.

Supongamos que el estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{while}(b)\{S_1\}S))$ necesita $k+1$ pasos en la semántica modificada para "dar el salto" al if. Aplicando la regla $\left[\text{while}_{\text{C}}^{\text{ALT}}\right]$ llegamos al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathbb{B}}\, \mathbf{while}(b')\{S_1\}S))$. Por otro lado, en la semántica original, el while se ve como un salto inmediato al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{if}(b)\{S_1; \mathbf{while}(b)\{S_1\}\}S))$ sobre el que ahora se aplica la misma ejecución de la expresión booleana anterior (por ser la misma expresión b) y llegamos al estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S_{\mathbb{B}}\, \mathbf{if}(b')\{S_1; \mathbf{while}(b)\{S_1\}\}S))$. Ahora, aplicando la hipótesis de inducción, tenemos

que ambos estados llegan al mismo estado en k pasos con lo que concatenando con el paso anterior tenemos que la propiedad se mantiene.

Una vez tenemos resueltas estas cuestiones, la equivalencia en la bisimulación entre CABS y ABS se sigue de un argumento completamente análogo al del caso del if.

Opción alternativa.

Otro modo de concluir con el caso del while sería crear un azucar sintáctico para el while de ABS que permitiera agrupar en su sintaxis el código de las asignaciones para el cómputo de la condición. De este modo, se podría modificar la regla del while para que actuara sobre esta construcción auxiliar y pudiera realizar el mismo paso que la semántica de CABS original para llegar a un equivalente a la vez. Esta opción sería equivalente a la anterior pero por simplicidad se deja solo comentada.

Caso thread

El caso del **thread** vuelve a traer consigo la discusión que ya habíamos tenido con las llamadas a funciones con parámetros. Las funciones llamadas en paralelo con **thread** pueden verse de nuevo como un operador, con la diferencia de que no tienen un resultado en su retorno. De nuevo sería necesario analizar el proceso de la bisimulación para las funciones con n-paramétros (con n variable) comprobando que se mantiene la equivalencia entre estados en cada uno de los pasos de la evaluación de los parámetros y, finalmente, en la creación final de los procesos.

Por simplicidad en este trabajo hablaremos del caso de las funciones sin parámetros, puesto que los restantes casos se reducen a comprobar que la evaluación de expresiones aritmético-lógicas se sigue comportando bien al usarse como paso por valor en funciones en lugar de en asignaciones, como ya hemos detallado anteriormente.

Sean, por tanto, los estados equivalentes $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, \mathbf{thread}\,\mathbf{func}(); S))$ de CABS y $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathrm{Int} f\ aux\_var\_k = new\ \mathrm{Imp} f(globalval); aux\_var\_k!f(); S_{\mathbf{ABS}}, t), t, \mathbf{attr})$, de ABS.

En la semántica de CABS, podemos usar la regla $\left[\mathrm{thread}^2_{\mathrm{C}}\right]$ que en un paso nos lleva al estado $(\mathbf{G}, \mathbf{F}, (\mathbf{RP} \cup (\mathbf{nil} : [], S_F)) \rightsquigarrow (local : s, S)$ con lo que ya habríamos terminado con la creación del proceso, donde S_F es el código de la función f que pasará a ejecutarse en paralelo.

En el caso de ABS, aplicando en un primer lugar la regla de creación de objetos llegaríamos a $(\mathbf{O} : (id', \mathrm{Imp} f, [], \bot, \mathbf{attr'}) \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\,[aux\_var\_k \mapsto id']\,, aux\_var\_k!f(); S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$. Desde este estado el único objeto capaz de mandar tareas al nuevo objeto creado es el identificado por id por tener en las variables locales de su tarea actual el identificador del nuevo objeto. Por tanto, la siguiente instrucción que se ejecuta, al aplicar la regla de paso de mensajes, nos lleva al estado $(\mathbf{O} : (id', \mathrm{Imp} f, (\mathbf{nil}, S^F_{\mathbf{ABS}}, t''), \bot, \mathbf{attr'}) \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\,[aux\_var\_k \mapsto id']\,, S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ con lo que llegaríamos a un estado equivalen-

te al que hemos llegado en la semántica de CABS.

Cabe mencionar que la razón por la que ambos ámbitos de variables locales se encuentran a nil es porque al realizar la llamada concurrente a una función sin argumentos esta no puede contar con más información de partida que la que se puede obtener de las variables globales. Para el caso de las llamadas a funciones con argumento las reglas semánticas garantizan que estos estarán disponibles en el ámbito local.

5.4. Conclusión de la correción

Con esto concluimos la corrección de la traducción dando un paso en la semántica de CABS. Ahora bien, el objetivo de este capítulo era conseguir ver que la evolución del programa escrito en CABS era equivalente a la que se lleva en cabo con su traducción en ABS. Esto se obtiene haciendo uso de una inducción sobre el número de pasos en la ejecución.

El caso base en el que damos cero pasos se mantiene trivialmente de partir de casos equivalentes. Para el caso $k + 1$, tenemos por hipótesis de inducción que se cumple la equivalencia de la ejecuciones de longitud menor que k y tenemos que dando un único paso podemos llegar a estados equivalentes desde los que quedan k pasos. Por tanto se sigue que concatenando los pasos de las ejecuciones, la propiedad se mantiene para $k+1$.

Capítulo 6

Implementación: compilador de CABS a ABS

Una vez llegados a este punto del trabajo, podemos quedar convencidos de lo interesante que puede ser contar con una implementación real de un compilador de CABS a ABS. Dicha implementación se encuentra disponible en un repositorio auxiliar alojado en `https://github.com/MaSteve/cabs`

En este capítulo, resumiremos el funcionamiento del compilador desarrollado comentando brevemente las librerías empleadas para este fin.

6.1. Introducción a los procesadores de lenguaje

La historia de los compiladores se remonta a los años 50 del siglo pasado. En aquella época, la compilación de programas consistía en sustituir las líneas de un lenguaje de alto nivel por el código de las rutinas a las que hacían referencia.

En torno a esos años y en la siguiente década, se establecieron las bases para la actual teoría de autómatas. Los lenguajes formales quedan clasificados por la jerarquía de Chomsky en función de su expresividad. Los primeros lenguajes de programación empiezan a surgir moviendose en el ámbito de las gramáticas incontextuales, que son aquellas que se pueden procesar con un autómata con pila.

Es en esta categoría donde podemos encontrar a la mayoría de lenguajes de programación. En concreto, se buscan aquellos subconjuntos de gramáticas en los que se puede garantizar que existe un reconocedor o autómata determinista. Algunos ejemplos de gramáticas son la $LL(k)$, la $LR(k)$ y la $LALR(k)$.

Puede decirse que la función actual de un compilador es la de traducir entre distintos lenguajes basandose en una teoría formal y preservando una cierta corrección entre los lenguajes de origen y destino. Las fases por las que pasa un compilador son las siguientes:

- Front-end o fase de análisis: que se compone a su vez en

  - Análisis léxico: a partir del código de origen se extraen los distintos elementos léxicos o tokens que lo componen, por ejemplo, las palabras reservadas del lenguaje, los nombres de variables, etc.
  - Análisis sintáctico: usando los token del análisis léxico, mediante el uso de un autómata determinista que reconozca la gramática del lenguaje, se crea un árbol de sintaxis abstracta en cuyos nodos podemos encontrar las piezas lógicas que componen nuestro programa, por ejemplo, las asignaciones de variables, la declaración de funciones, etc.
  - Análisis estático: en esta fase se analizan los identificadores usados en el programa, en busca de usos ilícitos como en el caso de las asignaciones en variables no declaradas, y los tipos de las expresiones que permiten descartar programas con errores. De este proceso se obtiene una tabla de símbolos que puede ser de ayuda en la traducción final del código.

- Back-end o fase de traducción: a partir del árbol de sintaxis abstracta y del resto de estructuras obtenidas en la etapa anterior, el compilador genera un código objeto entendible para la máquina o interprete para el que está destinado. En esta etapa también se pueden llevar a cabo algunas optimizaciones, pero hablar de ello no es nuestro objetivo.

6.2. JLex

JLex[1] es un generador de analizadores léxicos en java desarrollado en la Universidad de Princeton. A partir de una especificación de los tokens de un lenguaje, JLex genera un autómata capaz de reconocerlos recogido en una clase de Java.

El automata empleado en los analizadores léxicos es un autómata finito determinista, categoría en la que se encuentran los reconocedores de los lenguajes formales más básicos conocidos como lenguajes regulares.

La especificación del analizador léxico de CABS se puede encontrar en el archivo `parser.lex` dentro del paquete `parser`. A su vez, dentro de este paquete se encuentran la clase `Yytoken`, que será el tipo de objeto manipulado por el analizador sintáctico, y el archivo `Yylex.java`, con las clases asociadas al autómata finito determinista.

6.3. CUP

CUP (Construction of Useful Parsers) es un generador de analizadores sintácticos LALR(1) desarrollado en Java y mantenido por la Universidad Técnica de Munich (TUM).

Con una sintaxis similar a la de YACC, se puede especificar la gramática del lenguaje para el que uno quiere crear un parser y CUP genera un analizador recogido en una clase

[1] `https://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html`

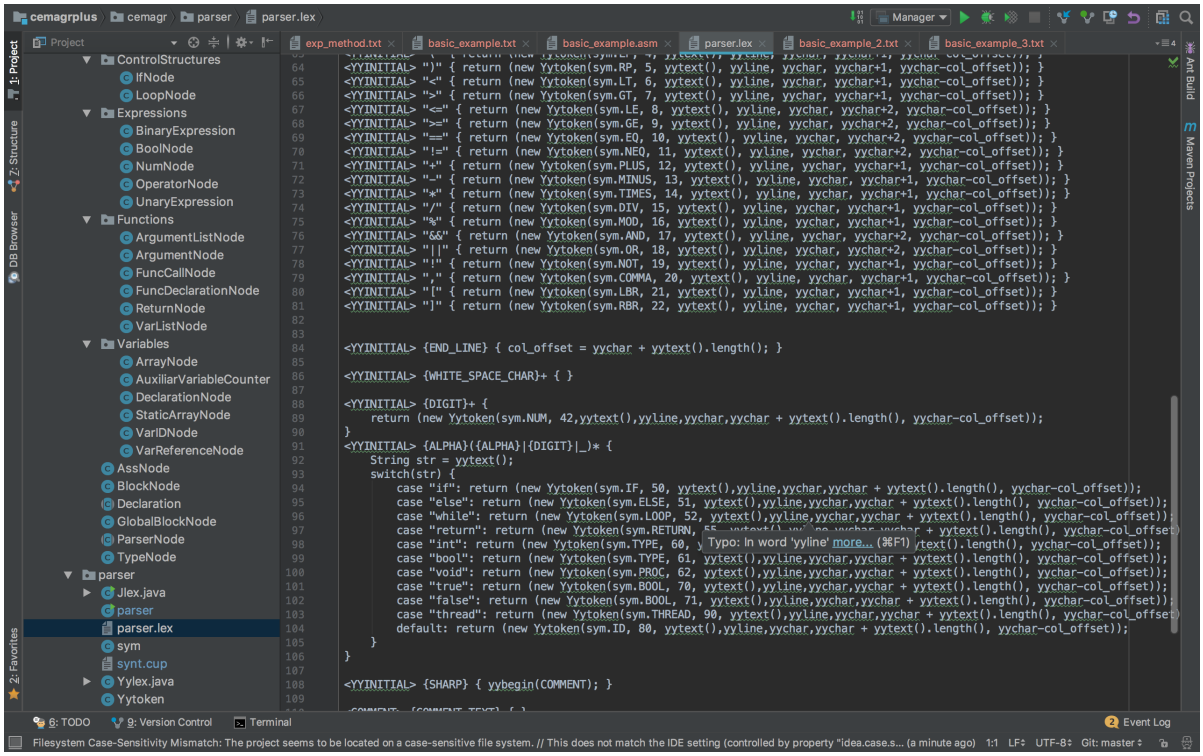

Figura 6.1: Captura de las reglas para la generación del lexer.

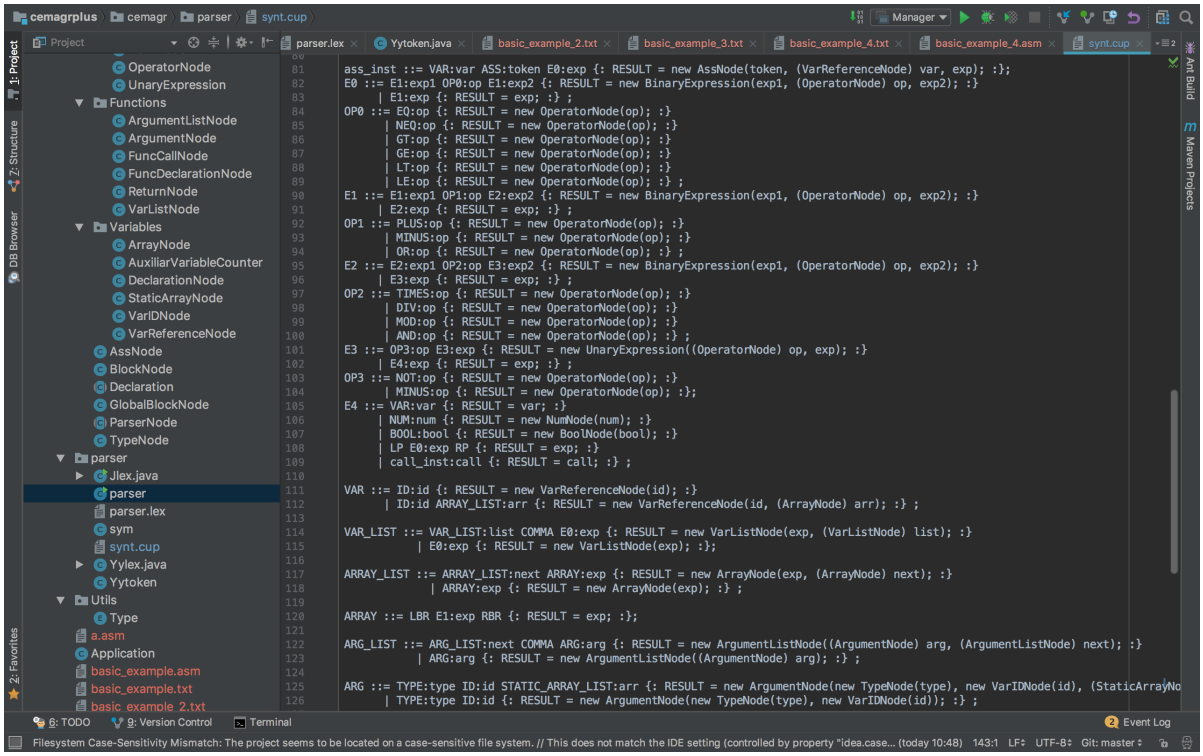

Figura 6.2: Captura de las reglas de la gramática de CABS.

de java.

La gramática de CABS se encuentra en el fichero `synt.cup` dentro del paquete `parser`. La clase `parser`, contenida también en el paquete anterior, es el archivo generado por CUP.

6.4. Estructura del compilador

El analizador sintáctico se apoya en gran medida en la estructura de nodos que el desarrollador del lenguaje determina para su creación. Puesto que tras la fase de parser se obtiene un primer árbol de sintaxis abstracta, es necesario ya para esta fase conocer cuáles son las clases empleadas para este proposito.

La jerquía de los nodos empleados en CABS se divide en los siguientes paquetes:

- ControlStructures: contiene los bloques asociados a las estructuras de decisión como son los condicionales (IfNode) y lo bucles (LoopNode).
- Expressions: contiene los nodos del árbol de sintaxis abstracta asociados a las expresiones aritmético-lógicas. Las clases pertenecientes a este paquete son las asociadas a los enteros y booleanos (NumNode y BoolNode respectivamente), los operadores (OperatorNode), las expresiones binarias de dos operandos y un operador (BinaryExpression) y unarias de un operando y un operador (UnaryExpression).
- Functions: paquete formado con todas las clases asociadas a la declaración de funciones y procedimientos y sus llamadas, incluidas las de creación de un nuevo hilo.
- Variables: paquete compuesto por las clases empleadas en la declaración de variables y arrays y su uso en asignaciones y expresiones aritmético-lógicas.

Además de estos nodos, existen otros nodos más generales que permiten completar los árboles de sintaxis abstracta con el resto de construcciones típicas de un lenguaje de programación. Algunos de estos nodos son las asignaciones (AssNode), los bloques de código (BlockNode y GlobalBlockNode) o los tipos de retorno y de variables (TypeNode).

La construcción del árbol de derivación se hace recursivamente, aprovechando la recursión propia de los analizadores sintácticos LALR. Posteriormente, el control del resto de fases es retomado por la clase `Manager`, que se encarga de iniciar los sucesivos recorridos del árbol para inicializar las tablas de identificadores de símbolos, es decir, asociar a cada uso de una variable o función su declaración, y realizar la comprobación de tipos y la existencia de la función *main*.

Por último, si llegados a este punto no hay ningún error de compilación, se procede a la generación de código, para lo cual, cada nodo cuenta con un método de traducción que propaga la llamada recursivamente a los nodos de los que depende. De este modo, aprovechando la estructura del árbol, se escribe de forma ordenada en un archivo las instrucciones del código destino.

Capítulo 7

SYCO: SYstematic testing tool for Concurrent Objects

SYCO es una de las herramientas implementadas sobre el lenguaje ABS que permite el testing de programas concurrentes escritos en este lenguaje. La idea de SYCO es que, a partir del código de un programa, el programador pueda saber de antemano las posibles ejecuciones del programa además de conocer si el programa presenta posibles situaciones de deathlock.

El núcleo de SYCO incluye implementaciones de técnicas de partial-order reduction que permiten la evaluación de ramas redundantes como las que se nos han presentado al principio de este trabajo en algunos ejemplos.

A través de una interfaz web, uno puede obtener visualmente el resultado que SYCO genera sobre el código proporcionado.

Veamos algunos ejemplos de uso sobre el código ABS generado a partir de un código en CABS.

```
1  int var;
2
3  int main() {
4      thread f(1);
5      thread f(2);
6      return 0;
7  }
8
9  void f(int value) {
10     var = value;
11 }
```

En este ejemplo tenemos que la función *main* lanza dos hilos que ejecutarán la función f dando como posibles resultados finales $var = 1$ y $var = 2$. La traducción a ABS quedaría, resumiendo parte de la traducción que no es necesaria para este ejemplo, como:

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

interface GLOBAL {
  Int getvar();
  Unit setvar(Int val);
  Unit initialize();
}

class GlobalVariables() implements GLOBAL {
  Int var = 0;
  Unit initialize() {
  }
  Int getvar() {
    return var;
  }
  Unit setvar(Int val) {
    var = val;
  }
}

interface Intf {
  Unit f(Int value);
}
interface Intmain {
  Int main();
}

class Impf(GLOBAL globalval) implements Intf {
  Unit f(Int value) {
    await globalval!setvar(value);
  }
}

class Impmain(GLOBAL globalval) implements Intmain {
  Int main() {
    Intf aux_var_0 = new Impf(globalval);
    Int aux_var_1 = 1;
    aux_var_0!f(aux_var_1);
    Intf aux_var_2 = new Impf(globalval);
    Int aux_var_3 = 2;
    aux_var_2!f(aux_var_3);
    Int aux_var_4 = 0;
    return aux_var_4;
  }
}
```

```
47
48 {
49   GLOBAL globalval = new GlobalVariables();
50   await globalval!initialize();
51   Intmain prog = new Impmain(globalval);
52   await prog!main();
53 }
```

Y el resultado que nos proporciona la herramienta SYCO es:

```
Independence constraints generated in 908 ms.
Number of executions: 2
Total time: 5
Total number of states explored during 2 executions: 16
Total number of tasks executed during 2 executions: 7
Execution 1, number of tasks: 6
(Click here to see the sequence diagram)
- State:
|------object(1,'GlobalVariables',[field(var,2)])
|------object(2,'Impmain',[field(globalval,ref(1))])
|------object(3,'Impf',[field(globalval,ref(1))])
|------object(4,'Impf',[field(globalval,ref(1))])
|------object(main,main,[])
- Trace: |------'Time: 0, Object: main, Task: 0:main'
|------'Time: 1, Object: GlobalVariables_1, Task: 1:initialize'
|------'Time: 2, Object: 0:main(54), Task: 0:main(54)'
|------'Time: 3, Object: Impmain_2, Task: 3:main'
|------'Time: 4, Object: 0:main(56), Task: 0:main(56)'
|------'Time: 5, Object: Impf_3, Task: 5:f'
|------'Time: 6, Object: GlobalVariables_1, Task: 7:setvar'
|------'Time: 7, Object: Impf_3, Task: 5:f(34)'
|------'Time: 8, Object: Impf_4, Task: 6:f'
|------'Time: 9, Object: GlobalVariables_1, Task: 9:setvar'
|------'Time: 10, Object: Impf_4, Task: 6:f(34)'
Execution 2, number of tasks: 6
(Click here to see the sequence diagram)
- State:
|------object(1,'GlobalVariables',[field(var,1)])
|------object(2,'Impmain',[field(globalval,ref(1))])
|------object(3,'Impf',[field(globalval,ref(1))])
|------object(4,'Impf',[field(globalval,ref(1))])
|------object(main,main,[])
- Trace: |------'Time: 0, Object: main, Task: 0:main'
|------'Time: 1, Object: GlobalVariables_1, Task: 1:initialize'
|------'Time: 2, Object: 0:main(54), Task: 0:main(54)'
|------'Time: 3, Object: Impmain_2, Task: 3:main'
|------'Time: 4, Object: 0:main(56), Task: 0:main(56)'
```

```
|------'Time: 5, Object: Impf_3, Task: 5:f'
|------'Time: 6, Object: Impf_4, Task: 6:f'
|------'Time: 7, Object: GlobalVariables_1, Task: 9:setvar'
|------'Time: 8, Object: GlobalVariables_1, Task: 7:setvar'
|------'Time: 9, Object: Impf_3, Task: 5:f(34)'
|------'Time: 10, Object: Impf_4, Task: 6:f(34)'
```

donde podemos ver las dos trazas obtenidas.

Para un ejemplo más elaborado como es

```
int var;
int var2;

int main() {
    thread f();
    thread g();
    return 0;
}

void f() {
    var = 1;
    var = 2;
}

void g() {
    var2 = 10 * var + var;
}
```

cuya traducción “resumida” es

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

interface GLOBAL {
  Int getvar();
  Unit setvar(Int val);
  Int getvar2();
  Unit setvar2(Int val);
  Unit initialize();
}
class GlobalVariables() implements GLOBAL {
  Int var = 0;
  Int var2 = 0;
  Unit initialize() {
  }
  Int getvar() {
    return var;
```

```
18    }
19    Unit setvar(Int val) {
20      var = val;
21    }
22    Int getvar2() {
23      return var2;
24    }
25    Unit setvar2(Int val) {
26      var2 = val;
27    }
28  }
29
30  interface Intf {
31    Unit f();
32  }
33
34  interface Intg {
35    Unit g();
36  }
37
38  interface Intmain {
39    Int main();
40  }
41
42  class Impf(GLOBAL globalval) implements Intf {
43    Unit f() {
44      Int aux_var_0 = 1;
45      await globalval!setvar(aux_var_0);
46      Int aux_var_1 = 2;
47      await globalval!setvar(aux_var_1);
48    }
49  }
50
51  class Impg(GLOBAL globalval) implements Intg {
52    Unit g() {
53      Int aux_var_2 = 10;
54      Int aux_var_3 = await globalval!getvar();
55      Int aux_var_4 = (aux_var_2 * aux_var_3);
56      Int aux_var_5 = await globalval!getvar();
57      Int aux_var_6 = (aux_var_4 + aux_var_5);
58      await globalval!setvar2(aux_var_6);
59    }
60  }
61
62  class Impmain(GLOBAL globalval) implements Intmain {
63    Int main() {
```

```
64      Intf aux_var_7 = new Impf(globalval);
65      aux_var_7!f();
66      Intg aux_var_8 = new Impg(globalval);
67      aux_var_8!g();
68      Int aux_var_9 = 0;
69      return aux_var_9;
70    }
71  }
72
73  {
74    GLOBAL globalval = new GlobalVariables();
75    await globalval!initialize();
76    Intmain prog = new Impmain(globalval);
77    await prog!main();
78  }
```

podemos rápidamente concluir cuales son los 6 posibles resultados finales de la ejecución del código: (AÑADIR!!!)

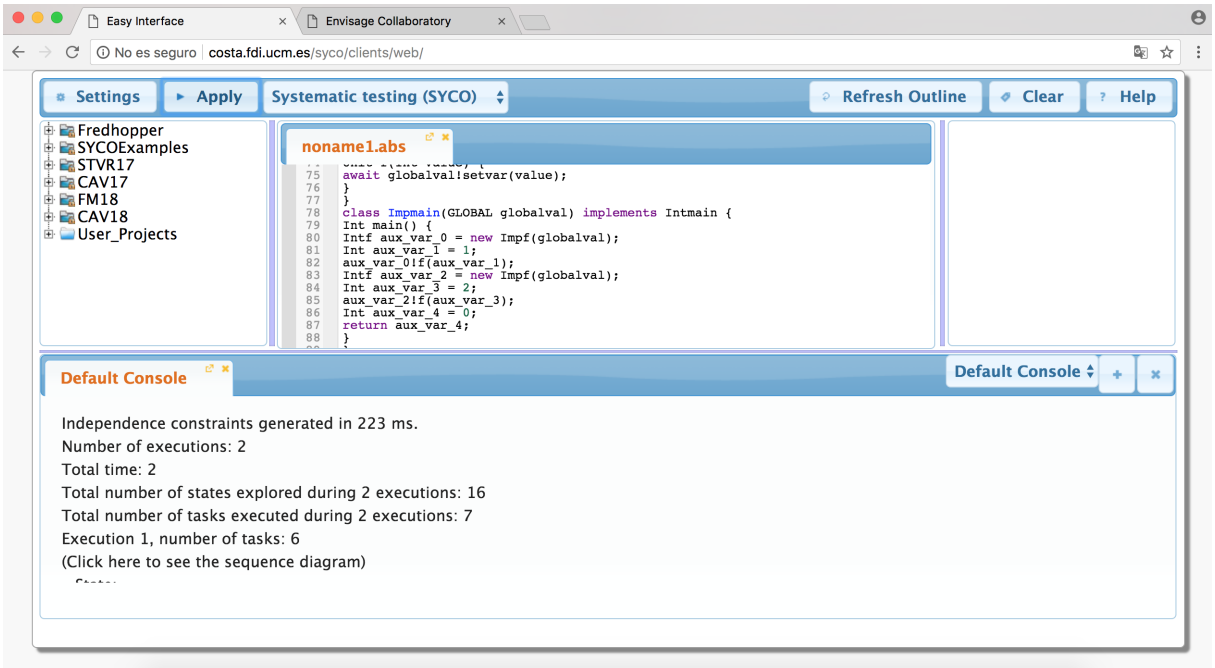

Figura 7.1: Captura de la herramienta SYCO.

Bibliografía

[1] Alicia Merayo Corcoba, *Detección estática de propiedades de ejecución de programas concurrentes con locks utilizando SACO*, Trabajo de Fin de Grado Matemáticas - Ingeniería Informática, dirigido por Elvira María Albert Albiol y Samir Genaim, 2017.

[2] Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. SDN-Actors: Modeling and Verification of SDN Programs. In FM 2018: 23th International Symposium on Formal Methods, Lecture Notes in Computer Science. Springer, 2018. To appear.