

Capítulo 1

Sintaxis y semántica (boceto)

1.1. Sintaxis de CABS (Still in development)

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

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

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

1.2. Semántica de CABS (faltan arrays!!)

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

1.2.1. Preámbulo semántico

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

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

Una primera parte de la representación del estado estará formada por las variables globales de nuestro programa. Definimos $\mathbf{G} = \mathbf{Var} \hookrightarrow \mathbb{V}$ el conjunto de funciones parciales del conjunto de variables al conjunto de valores. El estado actual de nuestras variables

globales será una función de este conjunto y por lo general nos referiremos a ella con la letra G . Posteriormente cuando hablemos de variables locales también las definiremos como un conjunto de funciones similares pero que trataremos por separado por comodidad.

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

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

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

La función **init**.

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

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

La función (regla) **start**.

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

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

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

Semántica (Expresiones aritmético-lógicas).

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

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

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

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

$$\begin{aligned}
& [\text{num}_{\mathcal{A}}] \frac{}{\langle n, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle \mathcal{N} \llbracket n \rrbracket, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle} \\
& [\text{var}_{\mathcal{A}}^L] \frac{\text{local}(x) = v \quad G(x) = \text{undef}}{\langle x, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S)) \rangle} \\
& [\text{var}_{\mathcal{A}}^G] \frac{G(x) = v \quad \text{local}(x) = \text{undef}}{\langle x, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S)) \rangle} \\
& [\odot_{\mathcal{A}}^1] \frac{\langle a_1, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle a'_1, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')) \rangle}{\langle a_1 \odot a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle a'_1 \odot a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')) \rangle} \\
& [\odot_{\mathcal{A}}^2] \frac{\langle a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle a'_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')) \rangle}{\langle v \odot a_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle v \odot a'_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')) \rangle} \\
& [\odot_{\mathcal{A}}^3] \frac{}{\langle v_1 \odot v_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle v_1 \odot_{\mathcal{N}} v_2, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle} \\
& [\text{unstack}_{\mathcal{A}}^1] \frac{\langle a, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle \rightarrow_{\mathbf{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_{\mathbf{Aexp}} \langle \text{UNSTACK}(a'), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s', S')) \rangle} \\
& [\text{unstack}_{\mathcal{A}}^2] \frac{}{\langle \text{UNSTACK}(v), (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S)) \rangle \rightarrow_{\mathbf{Aexp}} \langle v, (\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (s, S)) \rangle}
\end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

Capítulo 2

Traducción a ABS

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

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

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

2.1. Variables globales y funciones

La ausencia de memoria 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;
```

```

5 }
6
7 int main() {
8     thread f();
9     var1 = 1;
10    return 0;
11 }

```

En otras palabras, queremos que nuestra traducción a ABS pueda llegar a ambos interleavings, es decir, que el valor final de *var1* pueda ser 1 o 2.

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

```

1 module Cabs;
2 import * from ABS.StdLib;
3
4 interface Functions {
5     Unit f();
6     Unit main();
7 }
8
9 class Prog() implements Functions {
10     Int var1 = 0;
11
12     Unit f() {
13         var1 = 2;
14     }
15
16     Unit main() {
17         this!f();
18         var1 = 1;
19     }
20 }
21
22 {
23     Functions prog = new Prog();
24     prog.main();
25 }

```

Sin embargo, esta traducción nos lleva a un código ABS donde solo es posible uno de los dos interleavings (concretamente el que termina con *var1* valiendo 2).

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

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

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

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

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

```
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 tras el *await*. De nuevo solo conseguimos la mitad de los 4 estados finales posibles.

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

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

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

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

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

```
1 module Cabs;
2 import * from ABS.StdLib;
3
4 interface GLOBAL {
5     Int getvar1();
6     Unit setvar1(Int val);
7     Int getvar2();
8     Unit setvar2(Int val);
9 }
10
11 class GlobalVariables() implements GLOBAL {
12     Int var1 = 0;
13     Int var2 = 0;
14
15     Int getvar1() {
16         return var1;
17     }
18
19     Unit setvar1(Int val) {
20         var1 = val;
21     }
22
23     Int getvar2() {
24         return var2;
25     }
26
27     Unit setvar2(Int val) {
28         var2 = val;
29     }
30 }
31
32 interface Intf {
33     Unit f();
34 }
35
36 interface Intmain {
37     Int main();
38 }
39
40 class Impf(GLOBAL globalval) implements Intf {
41     Unit f() {
42         await globalval!setvar1(2);
43         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 }

```

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

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

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

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

2.2. Arrays locales y globales