Capítulo 1

Sintaxis y semántica (boceto)

1.1. Sintaxis de CABS (Still in development)

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

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

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

1.2. Semántica de CABS (faltan arrays!!)

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

1.2.1. Preámbulo semántico

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

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

Una primera parte de la representación del estado estará formada por las variables globales de nuestro programa. Definimos $G = \mathbf{Var} \hookrightarrow \mathbb{V}$ 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 $F = \mathbf{Func} \hookrightarrow (\mathbf{T} \times \mathbf{Stm} \times \mathbf{Args} \times (\mathbf{Exp} \cup \{\varepsilon\}))$ el conjunto de funciones parciales que asocia un nombre de función a su definición. Una función quedará definida por su tipo de retorno $t \in \mathbf{T}$, su código $S \in \mathbf{Stm}$, sus argumentos de entrada $\mathrm{arg} \in \mathbf{Args}$ y su expresión de retorno $e \in \mathbf{Exp} \cup \{\varepsilon\}$. Según las necesidades, la expresión de retorno será una expresión booleana, una expresión aritmética o simplemente será la expresión vacia, empleada para los procedimientos.

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

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

La función init.

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

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

La función (regla) start.

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

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

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

Semántica (Expresiones aritmético-lógicas).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Capítulo 2

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.

2.1. 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.

2.2. 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.

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

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

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

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

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

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

$$\frac{\mathcal{B}\,[\![b]\!]_{loc,\mathbf{attr}} = \textsc{true}}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}S, t), t, \mathbf{attr}), \mathbf{C}) \rightarrow (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S_1 S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\frac{\mathcal{B}\,[\![b]\!]_{loc,\mathbf{attr}} = \textsc{false}}{(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}S, t), t, \mathbf{attr}), \mathbf{C}) \rightarrow (\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S_2 S, t), t, \mathbf{attr}), \mathbf{C})}$$

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

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

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

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

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

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

2.3. 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 3

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.

3.1. Variables globales y funciones

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

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

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

```
int var1;

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

```
}

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

En otras palabras, queremos que nuestra traducción a ABS pueda llegar a ambos interleavings, es decir, que el valor final de $var1$ 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 $var1$ 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.

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

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

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

```
int var1;
int var2;

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

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

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

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

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

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

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

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

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

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

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

  Int getvar1() {
    return var1;
  }

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

  Int getvar2() {
    return var2;
  }

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

interface Intf {
  Unit f();
}

interface Intmain {
  Int main();
}

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

```

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

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

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

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

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

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

3.2. Arrays locales y globales

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

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

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

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

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

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

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

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

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

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

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

El código generado para el siguiente ejemplo

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

quedaría de este modo:

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

class GlobalVariables() implements GLOBAL {
  ArrayInt array;

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

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

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

...

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

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

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

```
 Unit init();
}

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

3.2.1. 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!!!!).

3.3. 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 **Aux** : $\mathbb{N} \to$ **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 $\mathcal{C}_{\mathbf{Aexp}} : \mathbf{Aexp} \rightarrow \mathbb{N} \rightarrow (\mathbf{ABS} \times \mathbb{N})$ y su homóloga $\mathcal{C}_{\mathbf{Bexp}}$ para las expresiones booleanas:

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

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.

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

3.4. 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 $\mathcal{C}_{arg} : \mathbf{Args} \rightarrow \mathbf{Args_{ABS}}$ que en esencia lo único que hace es cambiar los tipos de CABS a los de ABS. (Arrays!!!!)

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

Usando esta traducción de argumentos tenemos que

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

y para las variables globales tenemos

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

Capítulo 4

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.

4.1. 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 **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 **RP** se corresponde con un conjunto de tareas en los objetos de **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.

4.2. 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.

4.2.1. 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 $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc\left[x \mapsto \mathcal{A}\,[\![aux\_var\_k]\!]_{loc,\mathbf{attr}}\right], S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ 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 [$\mathrm{ass}^1_{\mathrm{C}}$] que delega la acción en las reglas para expresiones aritmético-lógicas. La regla aplicable en esta situación es [$\mathrm{num}_{\mathcal{A}}$] 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 $(\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, \mathrm{Intf}\ aux\_var\_(k-1) = new\,\mathrm{Impf}(globalval); Int\,aux\_var\_k = await\,auxi\_var\_(k-1)!f(); x = aux\_var\_k; S_{\mathbf{ABS}}, t), t, \mathbf{attr}), \mathbf{C})$ donde la variable auxiliar k-esima es la variable asignada a la expresión $f()$.

La semántica de CABS nos permite usar la regla [$\mathrm{ass}^1_{\mathrm{C}}$] con la regla [$\mathrm{call}_{\mathcal{A}}$] resultando en el estado $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathbf{nil} : local : s, S_F\, x = \textsc{unstack}\,(a)\,; S))$ 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

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

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

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

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 $[\mathrm{ass}^1_{\mathrm{C}}]$ con la regla $\left[\mathrm{unstack}^2_{\mathcal{A}}\right]$ 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

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

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

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

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

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

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

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

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

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

que son equivalentes.

Casos $x = f(a_1, \ldots, a_n)$ y $x =$ 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.

4.2.2. Variables globales (TODO!!!)

4.3. 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 $\left[\mathrm{Decl}_{\mathrm{C}}^{int}\right]$ o $\left[\mathrm{Decl}_{\mathrm{C}}^{bool}\right]$ 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:

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

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

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 $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathit{local} : s, \mathbf{while}(b)\{S_1\}S))$, 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 $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathit{local} : s, \mathbf{if}(\nu)\{S_1 \mathbf{while}(b)\{S_1\}\}; S))$, 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 $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathit{local} : s, \mathbf{while}(b)\{S_1\}S))$ 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 $(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (\mathit{local} : s, \mathbf{if}(b)\{S_1; \mathbf{while}(b)\{S_1\}\}S))$ 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