Estudio y desarrollo de técnicas para el testing de programas concurrentes

CABS: traducción de C a ABS

Marco Antonio Garrido Rojo[1]

21 de junio de 2018

[1] https://github.com/MaSteve/CABS-Slides

Introducción: historia y origen

- En un principio, la mayoría de sistemas solo podían ejecutar una única secuencia de instrucciones a la vez.
- Con el avance de los ordenadores durante las últimas décadas del siglo pasado, se permite la ejecución simultánea de varios hilos de ejecución.
- Objetivo: obtener un mejor rendimiento.
- Hoy en día es común el uso de la programación concurrente en la mayoría de aplicaciones y programas de todo ámbito, en especial, el relacionado con el cloud computing.

Introducción: inconvenientes

- La ejecución paralela conlleva riesgos adicionales no presentes en los programas secuenciales.
- El principal problema que presenta es la presencia de una memoria compartida sobre la que los distintos hilos realizan modificaciones en cualquier momento.
- Asociado a esto, pueden ocurrir deadlocks, carreras de datos o comportamientos impredecibles, además de los problemas de sincronización.

Introducción: soluciones

- Podemos evitar que ocurran estableciendo restricciones mediante el uso de semáforos y cerrojos o implementaciones de algoritmos de más bajo nivel como el tie-breaker.
- El uso de estos mecanismos puede ser bastante **complejo**.

Introducción: interleavings

- La existencia de los interleavings puede dar lugar a distintos resultados con el mismo programa.
- En este terreno se sigue investigando para encontrar un modo definitivo que permita solucionar los problemas o detectarlos.
- Uno de los primeros asuntos estudiados es determinar el estado o estados finales a los que se llega ejecutando un programa concurrente.

Introducción: interleavings

```
1  int var;
2
3  proc f1() {
4     var := 1;
5  }
6
7  proc f2() {
8     var := 2;
9  }
10
```

Introducción: interleavings

- Explosión exponencial en el número de interleavings en proporción al número de hilos y al número de instrucciones.
- No siempre hay una correlación con el número de estados finales.
- No todos los interleavings tienen la misma importancia y algunos de ellos son equivalentes entre sí.
- En este ámbito, se intentan llevar a cabo razonamientos formales de análisis, verificación y testing.
- DPOR (Dynamic partial order reduction): una de las técnicas de más éxito para hacer verificación y testing que permite detectar interleavings redundantes.

Introducción: interleavings

```
1  int var1;
2  int var2;
3
4  proc f1() {
5     var1 := 1;   # 1
6     var1 := 2;   # 2
7  }
8
9  proc f2() {
10    var2 := 2;   # 3
11    var2 := 1;   # 4
12 }
13
```

Introducción: grano

- ¿Cuál es la atomicidad?
  - Asignaciones atómicas: grano grueso.
  - ¿Lectura de valores?: grano fino.
  - ¡Secciones de código enteras!: . . .
- ¿Qué ocurre si no hay memoria compartida?

Introducción: actores

- Cada objeto ejecuta sus tareas de forma concurrente con respecto a las del resto de objetos.
- Una única tarea a la vez por objeto. El resto de tareas esperan en una cola cuyo orden no es determinable.
- El paso de mensajes indica qué método desea ejecutar un objeto (pudiendo ser uno propio o perteneciente a otro objeto).
- 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 es non-preemptive.

Introducción: ventajas

- Reducción notable del número de interleavings: importante cuando se aplican técnicas de testing sistemático.
- Presente en lenguajes como Erlang, Scala y ABS.
- En concreto, ABS cuenta con una amplia cantidad de herramientas, entre ellas, SYCO, que implementa DPOR para obtener los posibles resultados finales de un programa.

Introducción: objetivo

- Tenemos que los lenguajes más importantes tienen una concurrencia de grano fino y que, por otro lado, el uso de otro tipo de concurrencias ayuda en los razonamientos formales.
- ¿Sería posible pasar de un modelo de concurrencia a otro preservando el mismo comportamiento? Eso pretende este trabajo.
- Acercar las herramientas de ABS a un lenguaje que no sea de modelado, como C, aportando una implementación de una traducción y una demostración formal.
- Un lenguaje que permita recrear una concurrencia entre procesos a nivel de grano fino.
- Un lenguaje sencillo pero completo: **CABS**.

CABS: sintaxis

- Subconjunto de C con sintaxis para hebras.
- Tipos estáticos y arrays.

```
1 type global_var1;
2 type global_var2;
3
4 type nombre_de_funcion(args...) {
5   ...
6   codigo
7   ...
8   return exp
9 }
```

CABS: sintaxis

```
1  int array[10];
2  int global_var;
3
4  int f(int res, int arr[10]) {
5    int var;
6    var = 2 * res + arr[1];
7    return var;
8  }
9
10 int main() {
11   array[0] = 10;
12   array[1] = 5;
13   int res;
14   res = array[0] + 1;
15   global_var = f(res, array);
16   return 0;
17 }
```

CABS: sintaxis

```
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 }
```

CABS: semántica

- Estado: $(\mathbf{G}, \mathbf{F}, \mathbf{RP})$.
- Ámbito global de variables ($\mathbf{G}$): asocia nombres de variables a valores de $\mathbb{V}$.
- Definición de funciones: $\mathbf{F}(\mathbf{func}) = (t, S_F, args_F, a_{ret})$.
- Lista de marcos de ejecución: $\mathbf{RP} \rightsquigarrow (local : s, S)$

CABS: semántica

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

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

CABS: semántica de las **Aexp**

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

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

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

CABS: semántica de las **Aexp**

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

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

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

CABS: semántica de las **Aexp**

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

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

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

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

CABS: semántica de las asignaciones

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

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

$$[\text{ass}^3_C] \quad \frac{is\_global(var)}{(\mathbf{G}, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s,\ \text{var} = v; S)) \rightarrow (\mathbf{G}\,[\ \text{var} \mapsto v]\,, \mathbf{F}, \mathbf{RP} \rightsquigarrow (local : s, S))}$$

CABS: semántica del if

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

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

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

CABS: semántica del thread

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

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

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

CABS: ejemplo

```
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 }
```

CABS: ejemplo

Excluyendo algunos pasos:

$$(\mathbf{G}, \mathbf{F}, (\mathbf{nil}, \mathbf{thread}\, f(1); \mathbf{thread}\, f(2);)) \rightarrow$$
$$(\mathbf{G}, \mathbf{F}, (\mathbf{nil}\,[value \mapsto 1], var = value;) : (\mathbf{nil}, \mathbf{thread}\, f(2);)) \rightarrow$$
$$(\mathbf{G}, \mathbf{F}, (\mathbf{nil}\,[value \mapsto 1], var = value;) : (\mathbf{nil}\,[value \mapsto 2], var = value;) : (\mathbf{nil}, \varepsilon)) \rightarrow$$
$$(\mathbf{G}\,[var \mapsto 1], \mathbf{F}, (\mathbf{nil}\,[value \mapsto 1], \varepsilon) : (\mathbf{nil}\,[value \mapsto 2], var = value;) : (\mathbf{nil}, \varepsilon)) \rightarrow$$
$$(\mathbf{G}\,[var \mapsto 2], \mathbf{F}, (\mathbf{nil}\,[value \mapsto 1], \varepsilon) : (\mathbf{nil}\,[value \mapsto 2], \varepsilon) : (\mathbf{nil}, \varepsilon)) \rightarrow$$
$$(\mathbf{G}\,[var \mapsto 2], \mathbf{F}, [])$$

ABS: sintaxis

- Sintaxis parecida a la de Java (aproximadamente).
- Clases que implementan interfaces para exponer los métodos públicos.
- Tipos futuro para esperar resultados del paso de mensajes (await).

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

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

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

ABS: semántica

- Estado: $(\mathbf{O}, \mathbf{C})$.
- Definición de las clases: $\mathbf{C}\,[\![c]\!] = (\text{Inter}, attr_c, met, args_c)$.
- Lista de objetos instanciados (actores):
  $\mathbf{O} \rightsquigarrow (id, c, \mathbf{RT} \rightsquigarrow (loc, S, t), t, \mathbf{attr})$

ABS: algunas reglas semánticas

$$\left[\text{ass}^1_{\mathbf{ABS}}\right] \quad \frac{\mathit{is\_local}(x) \qquad \mathit{is\_Int}(x)}{(\mathbf{O} \rightsquigarrow (\mathit{id}, c, \mathbf{RT} \rightsquigarrow (\mathit{loc}, x = a; S, t), t, \mathbf{attr}), \mathbf{C}) \rightarrow (\mathbf{O} \rightsquigarrow (\mathit{id}, c, \mathbf{RT} \rightsquigarrow (\mathit{loc}\left[x \mapsto \mathcal{A}\left[\!\left[a\right]\!\right]_{\mathit{loc},\mathbf{attr}}\right], S, t), t, \mathbf{attr}), \mathbf{C})}$$

$$\left[\text{obj}_{\mathbf{ABS}}\right] \quad \frac{\mathbf{C}\left[\!\left[c'\right]\!\right] = (\text{Inter}, \mathit{attr}_c, \mathit{met}, \mathit{args}_c) \qquad \mathit{check\_args}(\mathit{args}_c, \mathit{args})}{(\mathbf{O} \rightsquigarrow (\mathit{id}, c, \mathbf{RT} \rightsquigarrow (\mathit{loc}, \text{Inter } \mathit{inter} = \mathit{new}\ c'(\mathit{args}); S, t), t, \mathbf{attr}), \mathbf{C}) \rightarrow (\mathbf{O} : o \rightsquigarrow (\mathit{id}, c, \mathbf{RT} \rightsquigarrow (\mathit{loc}\left[\mathit{inter} \mapsto \mathit{id}'\right], S, t), t, \mathbf{attr}), \mathbf{C})}$$

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

ABS: algunas reglas semánticas

$$\left[\text{tsk}^{\text{SYNC}}_{\textbf{ABS1}}\right] \quad \frac{loc \cup attr\ [\![int]\!] = id' \qquad \mathbf{C}\ [\![c']\!] = (\text{Inter}, attr_c, met, args_c) \qquad contains(met, m) \qquad check\_args(args_m, args)}{(\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT}', t', \mathbf{attr}')(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\ x = await\ int!m(args); S, t), t, \mathbf{attr}), \mathbf{C}) \rightarrow st}$$

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

$$\left[\text{ret}^{1}_{\textbf{ABS}}\right] \quad \frac{tsk = (loc', \varepsilon(\nu), t'')}{(\mathbf{O} \rightsquigarrow (id', c', \mathbf{RT}' \rightsquigarrow tsk, t', \mathbf{attr}')(id, c, \mathbf{RT} \rightsquigarrow (loc, Int\ x = await\ t'; S, t), t, \mathbf{attr}), \mathbf{C}) \rightarrow st}$$

donde $o' = (id', c', \mathbf{RT}', t', \mathbf{attr}')$ y $st = (\mathbf{O} \rightsquigarrow o'(id, c, \mathbf{RT} \rightsquigarrow (loc\,[x \mapsto \nu], S, t), t, \mathbf{attr}), \mathbf{C})$

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

ABS: ejemplo

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

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

  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 funcf1 = new Impf(globalval);
    funcf1!f(1);

    Intf funcf2 = new Impf(globalval);
    funcf2!f(2);

    return 0;
  }
}

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

Traducción

- Necesitamos crear de forma artificial un ámbito global de variables (memoria compartida ausente en ABS).
- La traducción de las funciones y procedimientos tienen que ser visibles desde cualquier punto del programa (¿métodos públicos de interfaz o métodos privados de una única clase?).
- La concurrencia tiene que ser de grano fino (ABS implementa el modelo de actores).
- Hay que dar un soporte a los arrays en ámbito local y global (ABS tiene un tipo de lista enlazada).

Traducción: ámbito global de variables y funciones

```
int var1;

void f() {
    var1 = 2;
}

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

Así solo es posible un interleaving.

Tal vez usando:

```
    this!f();
    suspend;

```

```
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();
}
```

Traducción: ámbito global de variables y funciones

Tampoco funciona la idea del *suspend*

```
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 }
```

Concluimos que cada función tiene que ir en un objeto distinto.

Traducción: ámbito global de variables y funciones

¿Qué hacemos ahora con las variables globales? Ya no pueden ser atributos de una clase que implemente una función. Necesita su propio objeto.

```
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;
  }
}
```

Usaremos llamadas asíncronas para manejar este objeto:

```
  globalval!setvar(1);
  Int x = await globalval!getvar();

```

Traducción: arrays locales y globales

El mismo argumento que hemos empleado para las variables globales (preservación de los interleavings) se puede aplicar a los arrays encapsulando una lista de ABS en una clase.

```
1  interface ArrayInt {
2    Int getV(Int indx);
3    Unit setV(Int indx, Int value);
4  }
5
6  class ArrayIntC(Int size) implements
       ArrayInt{
7    List<Int> list = copy(0, size);
8
9    Int getV(Int indx) {
10     return nth(this.list, indx);
11   }
12
13   Unit setV(Int indx, Int value) {
14     Int i = 0;
15     List<Int> prev = Nil;
16     List<Int> post = list;
17     while (i < indx) {
18       Int elem = head(post);
19       prev = appendright(prev, elem);
20       post = tail(post);
21       i = i + 1;
22     }
23     prev = appendright(prev, value);
24     post = tail(post);
25     this.list = concatenate(prev, post);
26   }
27 }
```

Para el ámbito global hay que añadir algunos métodos extra (*retrievearray*, *init* . . .).

Traducción: función de traducción **Aexp**

$$\mathcal{C}_{\mathbf{Aexp}}\,[\![n]\!]\,v = (Int\,(\mathbf{Aux}\,v) = n, v+1)$$
$$\mathcal{C}_{\mathbf{Aexp}}\,[\![x]\!]\,v = (Int\,(\mathbf{Aux}\,v) = x, v+1) \text{ donde } x \text{ es variable entera local.}$$
$$\mathcal{C}_{\mathbf{Aexp}}\,[\![x]\!]\,v = (Int\,(\mathbf{Aux}\,v) = await\ \text{globalval!get}x(), v+1)$$
$$\text{donde } x \text{ es variable entera global.}$$
$$\mathcal{C}_{\mathbf{Aexp}}\,[\![a_1 \odot_{\mathcal{A}} a_2]\!]\,v = (c_1; c_2; Int\,(\mathbf{Aux}\,v'') = (\mathbf{Aux}\,(v'-1)) \odot_{\mathcal{A}} (\mathbf{Aux}\,(v''-1)), v''+1)$$
$$\text{donde } \mathcal{C}_{\mathbf{Aexp}}\,[\![a_1]\!]\,v = (c_1, v') \text{ y } \mathcal{C}_{\mathbf{Aexp}}\,[\![a_2]\!]\,v' = (c_2, v'')$$

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

Traducción: función de traducción del if y del while

$$\mathcal{C}\,[\![\mathbf{if}(b)\{S_1\}\mathbf{else}\{S_2\}]\!]\,v = (c_1; \mathbf{if}(\mathbf{Aux}\,(v'-1))\{c_2\}\mathbf{else}\{c_3\}, v''')$$
$$\text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v = (c_1, v'),\ \mathcal{C}\,[\![S_1]\!]\,v' = (c_2, v'') \text{ y } \mathcal{C}\,[\![S_2]\!]\,v'' = (c_3, v''')$$
$$\mathcal{C}\,[\![\mathbf{while}(b)\{S\}]\!]\,v = (c_1; \mathbf{while}(\mathbf{Aux}\,(v'-1))\{c_2 c_3\,\mathbf{Aux}\,(v'-1) = \mathbf{Aux}\,(v'''-1);\}, v''')$$
$$\text{donde } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v = (c_1, v'),\ \mathcal{C}\,[\![S]\!]\,v' = (c_2, v'') \text{ y } \mathcal{C}_{\mathbf{Bexp}}\,[\![b]\!]\,v'' = (c_3, v''')$$

Traducción: función de traducción de una función y del thread

$$\mathcal{C}\,[\![int\,\mathbf{func}(arg)\{S\,\mathbf{return}\ a\}]\!]\,v = (\text{interface Int}\mathbf{func}\{Int\,\mathbf{func}(\mathcal{C}_{arg}\,arg);\}$$
$$\text{class Imp}\mathbf{func}(\text{GLOBAL globalval}) \text{ implements Int}\mathbf{func}\{$$
$$Int\,func(\mathcal{C}_{arg}\,arg)\{c\,c'\,\mathbf{return}\ \mathbf{Aux}\,(v''-1)\}\},v'')$$
$$\text{donde } \mathcal{C}\,[\![S]\!]\,v = (c,v') \text{ y } \mathcal{C}_{\mathbf{Aexp}}\,[\![a]\!]\,v' = (c',v'')$$

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

Corrección

- Se trata de ver que existe una equivalencia semántica entre el programa en CABS y su traducción a ABS.
- Ambas semánticas son de paso corto luego la idea propuesta pasa por hacer una bisimulación en ambas semánticas.
- Partiendo de dos estados equivalentes, uno de CABS y otro de ABS, hay que ver que un paso en una de las semánticas se corresponde con un paso en la otra semántica (obviando transiciones que no afectan al significado).
- Dándose esto para un paso, por inducción en la longitud de la secuencia de derivación, tenemos que se cumple para ejecuciones arbitrariamente largas.