UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Máster en Ingeniería y Tecnología de Sistemas Software

Trabajo Final de Máster

Depuración Reversible en Erlang

Autor:
Juan José González Abril

Tutor académico:
Germán Francisco Vidal Oriola

Fecha de lectura: __ de Septiembre de 2020
Curso académico 2019/2020

Resumen

En este proyecto hemos diseñado e implementado una nueva versión del depurador CauDEr, un depurador reversible para programas Erlang. La versión original del depurador consideraba programas en formato Core Erlang. Aunque esto simplificó la definición e implementación del depurador, también planteó una serie de problemas. Más concretamente, el depurador original muestra el avance de la ejecución mediante la representación, en formato Core Erlang, de la expresión que se está reduciendo en cada paso. Esta expresión, sin embargo, puede ser muy diferente del código fuente que escribió el programador, por lo que resulta confusa. Por ello, en este trabajo hemos definido la semántica reversible al nivel de Erlang (en vez de Core Erlang) y hemos reimplementado el depurador en base a la nueva semántica. Además, se ha rediseñado la interfaz de usuario para que muestre el código fuente de forma que la línea que se está ejecutando aparezca resaltada (en lugar de mostrar una expresión Core Erlang). Todo ello ha mejorado notablemente la facilidad de uso del depurador.

Palabras clave

Erlang, Depuración reversible, Consistencia causal

Keywords

Erlang, Reversible debugging, Causal consistency

Índice general

Capítulo 1

Introducción

La programación concurrente ha ganado popularidad en los últimos años. Sin embargo, se trata de una actividad difícil y propensa a errores, ya que la concurrencia permite comportamientos erróneos, como *deadlocks* y *livelocks*, que son difíciles de evitar, detectar y corregir. Una de las razones de estas dificultades es que estos comportamientos pueden aparecer solo en algunas circunstancias extremadamente raras.

La mayoría de los enfoques para la validación y depuración de software en lenguajes concurrentes basados en paso de mensajes como Erlang se basan en alguna forma de análisis o pruebas estáticas. Sin embargo, estas técnicas son útiles solo para encontrar algunas categorías específicas de problemas. Por otro lado, los depuradores tradicionales (como el incluido en la distribución Erlang/OTP) a veces no son particularmente útiles cuando un *interleaving* inusual genera un error, ya que recompilar el programa para su depuración y la realización de una nueva ejecución puede dar lugar a un comportamiento completamente diferente. En este contexto, la *depuración reversible causalmente consistente* puede ser útil para complementar los enfoques anteriores. Aquí, se puede ejecutar un programa en el depurador de manera controlada. Si aparece algo (potencialmente) incorrecto, el usuario puede detener el cálculo hacia adelante y retroceder (de una manera causalmente consistente), para buscar el origen del problema. En este contexto, decimos que un paso atrás es causalmente consistente si una acción no se puede deshacer hasta que todas las acciones que dependen de ella ya se hayan deshecho. La reversibilidad causalmente consistente es particularmente relevante para la depuración porque nos permite deshacer las acciones de un proceso paso a paso mientras ignoramos las acciones de los procesos restantes, a menos que estén relacionadas causalmente. En un depurador reversible tradicional, solo se puede retroceder exactamente en el orden inverso de la ejecución hacia adelante, lo que hace que centrarse en deshacer las acciones de un proceso dado sea mucho más difícil, ya que se pueden intercalar con acciones completamente independientes de otros procesos.

La motivación detrás de la actualización del depurador CauDEr, viene dada por la dificultad que puede suponer trabajar con Core Erlang durante la depuración de un programa, ya que a medida que crece el tamaño del programa se vuelve más complicado de entender. Además, en la última versión de la distribución Erlang/OTP, la versión 23, se produjo un cambio en el compilador para añadir soporte a nuevas características del lenguaje. Sin embargo, esto

conllevaba cambios en el código Core Erlang que se genera; concretamente las expresiones *receive* pasan a convertirse en una serie de operaciones primitivas, lo que obligaría a reescribir buena parte del depurador. Otra razón por la que se ha decidido actualizar el depurador ha sido para mejorar la interfaz de usuario y para poder mostrar todo el código fuente en lugar de unicamente la expresión a evaluar.

Es por todo esto que se ha decidido pasar a usar el *abstract format* de Erlang para trabajar con el código fuente durante las sesiones de depuración; además, al tratarse de una representación equivalente al código fuente, es trivial la trasformación de código fuente a *abstract format* y viceversa.

```
1 -module(factorial).
2 -export([fact/1]).
3
4 fact(0)             -> 1;
5 fact(N) when N > 0 -> N * fact(N - 1).
```

Figura 1.1: Ejemplo de un programa escrito en Erlang

```
1  module 'factorial' ['fact'/1]
2  'fact'/1 =
3    fun (_0) ->
4      case _0 of
5        <0> when 'true' -> 1
6        <N> when call 'erlang':'>'(N, 0) ->
7            let <_1> = call 'erlang':'-'(N, 1)
8            in let <_2> = apply 'fact'/1(_1)
9            in call 'erlang':'*'(N, _2)
10       <_3> when 'true' ->
11           primop 'match_fail'({'function_clause',_3})
12     end
13 end
```

Figura 1.2: Programa de la Figura 1.1 compilado a Core Erlang

En la Figura 1.1 se muestra un ejemplo del *abstract format* en un programa escrito en Erlang, y en la Figura 1.2 se muestra el mismo programa pero compilado a Core Erlang. Aquí se puede observar la gran diferencia que existe entre la representación en *abstract format* (código fuente) y en Core Erlang.

Capítulo 2

El Lenguaje Erlang

Erlang es un lenguaje de programación de propósito general, funcional y concurrente basado en paso de mensajes, el cual sigue principalmente el modelo de actores.

En este trabajo, por simplicidad, consideramos un subconjunto representativo de la sintaxis del lenguaje. En este capítulo introducimos la sintaxis y la semántica del subconjunto de Erlang considerado.

2.1. Sintaxis

La sintaxis del lenguaje se muestra en la Figura 2.1, donde se hace uso de la notación de Backus-Naur extendida (EBNF), que aparte de las construcciones estándar, hace uso de las construcciones $[\ldots]$ y $\{\ldots\}$, para indicar que un elemento es opcional y qué elemento se repite cero o más veces, respectivamente. En esta definición los elementos terminales usan un tipo de letra Courier y están delimitados por comillas simples y los no terminales se muestran en cursiva.

Podemos distinguir entre *expresiones*, *patrones* y *valores*. Los patrones se construyen a partir de variables, literales (valores atómicos), listas y tuplas. Por otro lado, los valores se construyen a partir de literales, listas y tuplas, es decir, son patrones básicos (sin variables). Las expresiones se denotan por $e, e', e_1, \ldots$, los patrones por $pat, pat', pat_1, \ldots$, y los valores por $v, v', v_1, \ldots$.

Un programa está formado por un conjunto de módulos, donde cada módulo está formado por una secuencia de definiciones de funciones. Cada función a su vez está definida por una o más clausulas con la forma $f(pat_1, \ldots, pat_n) \rightarrow e_1, \ldots, e_m$, donde f es un átomo. Como se puede observar, las clausulas pueden tener una secuencia de expresiones en su parte derecha. Las clausulas se prueban en orden, de arriba a abajo, usando emparejamiento de patrones (*pattern matching*). Además, las clausulas pueden tener una guarda (prefijada por la palabra reservada `when`) que deberá evaluarse a `true` para que la regla sea aplicable. Las guardas están agrupadas en secuencias separadas por un punto y coma, las cuales se evalúan de forma secuencial hasta que alguna sea `true`. Cada una de estas guardas a su vez están formadas por una secuencia de expresiones de guarda, donde cada expresión deberá evaluarse a `true` para que se cumpla la guarda. Las expresiones de guarda solo pueden estar compuestas por

program ::= *mod* {*mod*}
mod ::= *fun_def* {*fun_def*}
fun_def ::= *fun_clause* {‘;’ *fun_clause*} ‘.’
fun_clause ::= *Atom*‘(’[*pattern*]‘)’ [‘when’ *guard_seq*] ‘→’ *exprs*
fun_expr ::= ‘fun’ *fun* {‘;’ *fun*} ‘end’
fun ::= ‘(’[*pattern*]‘)’ [‘when’ *guard_seq*] ‘→’ *exprs*
patterns ::= *pattern* {‘,’ *pattern*}
pattern ::= *atomic* | *Var* | ‘{’[*patterns*]‘}’ | ‘[’[*patterns* [‘|’ *pattern*]]‘]’ | *unary_op pattern* | *pattern binary_op pattern*
exprs ::= *expr* {‘,’ *expr*}
expr ::= *atomic* | *Var* | ‘{’[*exprs*]‘}’ | ‘[’[*exprs* [‘|’ *expr*]]‘]’ | ‘if’ *if_clauses* ‘end’ | ‘case’ *expr* ‘of’ *cr_clauses* ‘end’ | ‘receive’ *cr_clauses* ‘end’ | *expr* ‘!’ *expr* | *pattern* ‘=’ *expr* | *unary_op expr* | *expr binary_op expr* | [*expr*‘:’]*expr*‘(’[*exprs*]‘)’ | *fun_expr*
atomic ::= *Atom* | *Char* | *Float* | *Integer* | *String*
unary_op ::= ‘+’ | ‘-’ | ‘not’ | ‘bnot’
binary_op ::= ‘==’ | ‘/=’ | ‘=<’ | ‘<’ | ‘>=’ | ‘>’ | ‘=:=’ | ‘=/=’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘div’ | ‘rem’ | ‘++’ | ‘--’ | ‘not’ | ‘and’ | ‘or’ | ‘xor’ | ‘andalso’ | ‘orelse’ | ‘bnot’ | ‘band’ | ‘bor’ | ‘bxor’ | ‘bsl’ | ‘bsr’
if_clauses ::= *if_clause* {‘;’ *if_clause*}
if_clause ::= *guard_seq* ‘→’ *exprs*
cr_clauses ::= *cr_clause* {‘;’ *cr_clause*}
cr_clause ::= *pattern* [‘when’ *guard_seq*] ‘→’ *exprs*
guard_seq ::= *guards* {‘;’ *guards*}
guards ::= *guard* {‘,’ *guard*}
guard ::= *atomic* | *Var* | ‘{’[*guards*]‘}’ | ‘[’[*guards* [‘|’ *guard*]]‘]’ | *unary_op guard* | *guard binary_op guard* | [‘erlang:’]*Atom*‘(’[*exprs*]‘)’

Figura 2.1: Reglas de sintaxis del lenguaje

valores atómicos, variables, tuplas, listas y llamadas a funciones predefinidas[1]. En general, una expresión puede incluir:

- Valores atómicos, los cuales pueden ser:
  - Átomos (es decir, constantes con un nombre), los cuales deben escribirse entre comillas simples (') si no comienza con una letra minúscula o si contiene otros caracteres además de los alfanuméricos, guión bajo (_) o @, por ejemplo, `hello` o `'World'`.
  - Caracteres, los cuales están prefijados por el signo del dolar ($), por ejemplo, `$a`.
  - Números enteros, por ejemplo, `42`.
  - Números en coma flotante, por ejemplo, `2.3` o `2.3e3`.

[1] La lista de expresiones permitidas en guardas se puede consultar en `https://erlang.org/doc/reference_manual/expressions.html#guard-expressions`

    - Cadenas, las cuales está delimitadas por comillas dobles ("), por ejemplo, `"CauDEr"`.

- Variables, las cuales empiezan con letra mayúscula, por ejemplo, `Nombre`.
- Tuplas, de la forma $\{e_1, \ldots, e_n\}$, donde tenemos una secuencia de expresiones, posiblemente vacía, delimitada por un par de llaves, por ejemplo, `{1,2,3,4,5}`.
- Listas, con una notación similar a la de Prolog, donde [] representa la lista vacía y $[e_1|e_2]$ es una lista donde e_1 es la cabeza de la lista y e_2 es la cola, por ejemplo, `[1,2,3|[4,5]]`.
- Expresiones *if*, que consisten en una serie de ramas compuestas por una guarda y una secuencia de expresiones (cuerpo); en una expresión *if* se evalúa el cuerpo de la primera rama cuya guarda se evalúe a `true`.
- Expresiones *case*, que consisten en una expresión y una serie de ramas compuestas por un patrón, una guarda opcional y una secuencia de expresiones (cuerpo); en este caso solo se evalúa el cuerpo de la primera rama cuyo patrón empareje con la expresión dada y cuya guarda se evalúe a `true`.
- Expresiones *receive*, que son prácticamente iguales que las expresiones *case* con la única diferencia que, en vez que realizar emparejamiento de patrones con una expresión especificada, se realiza con los mensajes enviados (en orden) a este proceso.
- Envío de mensajes, con la forma e_1 ! e_2, donde e_1 es el *pid* del proceso al cual se envía el mensaje y e_2 es el mensaje a enviar, por ejemplo, `Server ! ping`.
- Emparejamiento de patrones, con la forma $pat = e$, donde pat deber ser un patrón válido y e puede ser una expresión cualquiera, por ejemplo, `[A,B,C] = [1|[2,3]]`.
- Aplicación de funciones, de la forma $e_m{:}e_f(e_1, \ldots, e_n)$, donde e_m es el módulo al que pertenece la función, e_f es el nombre de la función y por último los argumentos que están delimitados por paréntesis y consisten en una secuencia, posiblemente vacía, de expresiones, por ejemplo, `erlang:length([1,2,3])`.
- Funciones anónimas, que son similares a la declaración de una función normal solo que en este caso no tienen nombre y tienen el prefijo `fun` y el sufijo `end`, por ejemplo, `fun(X) -> X * 2 end`
- Los típicos operadores aritméticos[2], relacionales[3] y booleanos[4], por ejemplo, `4/2`, `1==1.0`, `not true`, etc.

[2] `https://erlang.org/doc/reference_manual/expressions.html#arithmetic-expressions`

[3] `https://erlang.org/doc/reference_manual/expressions.html#term-comparisons`

[4] `https://erlang.org/doc/reference_manual/expressions.html#boolean-expressions`

Erlang incluye un cierto número de funciones *built-in* (BIFs); en este trabajo solo se han considerado las funciones sin efectos laterales, con la excepción de las funciones `self/0`, la cual devuelve el *pid* del proceso actual, `spawn/1` y `spawn/3`, que crean nuevos procesos (ver más adelante), y `send/2`, que realiza la misma función que el operador `'!'`, es decir el envío de mensajes.

```
1  -module(client_server).
2  -export([main/0, server/0, proxy/0]).
3
4  main() ->
5    S = spawn(?MODULE, server, []),
6    P = spawn(?MODULE, proxy, []),
7    client(P, S).
8
9  server() ->
10   receive
11     {C, N} ->
12       receive
13         M -> C ! N + M
14       end;
15     E -> error
16   end.
17
18 proxy() ->
19   receive
20     {T, M} -> T ! M
21   end.
22
23 client(P, S) ->
24   P ! {S, {self(), 40}},
25   S ! 2,
26   receive
27     N -> N
28   end.
```

Figura 2.2: Ejemplo de un programa cliente/servidor

Ejemplo 1. El programa de la Figura 2.2 implementa un esquema simple cliente/servidor con un servidor, un cliente y un proxy. La ejecución empieza con la llamada a la función `main/0`. Esta llamada crea el servidor y el proxy, suspendiendo entonces la ejecución a la espera de que lleguen mensajes. El cliente realiza dos peticiones $\{C, 40\}$ y 2, donde C es el *pid* del cliente (obtenido usando `self()`). La segunda petición va directamente al servidor, pero la primera se envía a través del proxy (que simplemente reenvía los mensajes que recibe), de modo que el cliente en realidad envía $\{S, \{C, 40\}\}$, donde S es el *pid* del servidor. Aquí, se espera que el servidor primero reciba el mensaje $\{C, 40\}$ y, después, 2, enviando por tanto de vuelta 42 al cliente C. Si el mensaje no tiene la estructura correcta, la rama *catch-all* "$E \rightarrow error$" devuelve *error* y detiene la ejecución.

2.2. Semántica

Al igual que [3], la semántica operacional se define mediante varias capas. La primera capa, el nivel más bajo, considera la evaluación de expresiones. Aquí,

distinguimos las expresiones secuenciales de las relacionadas con la concurrencia. El subconjunto secuencial del lenguaje es un típico lenguaje funcional impaciente de orden superior.

2.2.1. Evaluación de expresiones secuenciales

Vamos a comenzar por introducir primero algunas nociones básicas. Una *sustitución* θ es la aplicación de variables a expresiones, donde $\mathcal{D}om(\theta) = \{X \in Var \mid X \neq \theta(X)\}$ es su dominio. Las sustituciones se denotan normalmente por un conjunto (finito) de pares, por ejemplo, $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. La sustitución identidad se denota por *id*. La composición de sustituciones se denota mediante yuxtaposición, por ejemplo, $\theta\theta'$ denota una sustitución θ'' tal que $\theta''(X) = \theta'(\theta(X))$ para todo $X \in Var$. Usamos una notación sufija para la aplicación de sustituciones: dada una expresión e y una sustitución σ, la aplicación $\sigma(e)$ se denota por $e\sigma$.

A veces se usa $\overline{o_n}$ para denotar una secuencia de objetos sintácticos $o_1, \ldots, o_n$ para una cierta n. También se usa $\overline{o}$ cuando el número de objetos no es relevante.

A continuación, consideramos que cada expresión se puede escribir como $C[e]$, donde e es el siguiente *redex* a reducir de acuerdo con la semántica impaciente típica, e.g, una expresión como `case f(42) of ... end` se puede representar como C[`f(42)`] ya que la llamada `f(42)` es necesaria para evaluar la expresión *case*.

Las reglas para expresiones secuenciales se muestran en la Figura 2.3. Esta semántica se define usando triplas θ, e, S, donde θ es el entorno actual que almacena los valores de las variables del programa, e es la secuencia de expresiones a evaluar, y S es una pila.

A continuación se introducen brevemente las reglas de la semántica:

- La regla *Var* evalúa una variable simplemente buscando su valor en el entorno.

- La regla *Seq1* consume el valor de la secuencia, de modo que el resto de elementos puedan ser reducidos. En general, varias expresiones de Erlang se pueden reducir a una secuencia. Por desgracia, esto puede dar lugar a una expresión que es sintácticamente ilegal; por ejemplo, consideremos que la evaluación de la expresión `case f(42) of ... end` reduce `f(42)` a `X=42,X`. Entonces, nos quedaría la expresión `case X=42,X of ... end`, la cual no es sintácticamente correcta (el argumento de una expresión *case* debe ser una única expresión, las secuencias no están permitidas). Para solucionar este problema, las reglas *If* y *Case* almacenan el contexto actual en la pila hasta que la expresión se reduce a una única expresión; entonces, el contexto se recupera mediante la regla *Seq2*.

- La regla *If* comprueba las ramas en orden usando la función auxiliar `match_if`, la cual devuelve el cuerpo de la primera rama cuya guarda se evalúa a `true`. Como se menciona arriba, se almacena el contexto actual en la pila. Si ninguna guarda se evalúa a `true`, se genera un error `if_clause`.

- La regla *Case* empareja el argumento del *case*, v, con las ramas del *case* (posiblemente incluyendo guardas) usando la función auxiliar `match_case` y devuelve un par $\langle \theta_i, e_i \rangle$ con la sustitución correspondiente y el cuerpo de

$$(Var)\quad \frac{}{\theta, C[X], S \xrightarrow{\tau} \theta, C[\theta(X)], S}$$

$$(Seq1)\quad \frac{}{\theta, C[v, \overline{e}], S \xrightarrow{\tau} \theta, C[\overline{e}], S}$$

$$(Seq2)\quad \frac{}{\theta, v, C[\_]:S \xrightarrow{\tau} \theta, C[v], S}$$

$$(If)\quad \frac{\texttt{match\_if}(\overline{cl_n\theta}) = \overline{e}}{\theta, C[\texttt{if } cl_1; \ldots; cl_n \texttt{ end}], S \xrightarrow{\tau} \theta, \overline{e}, C[\_]:S}$$

$$(Case)\quad \frac{\texttt{match\_case}(v, \overline{cl_n\theta}) = \langle \theta', \overline{e} \rangle}{\theta, C[\texttt{case } v \texttt{ of } cl_1; \ldots; cl_n \texttt{ end}], S \xrightarrow{\tau} \theta\theta', \overline{e}, C[\_]:S}$$

$$(Match)\quad \frac{\texttt{match}(pat\theta, v) = \sigma}{\theta, C[pat = v], S \xrightarrow{\tau} \theta\sigma, C[v], S}$$

$$(Op)\quad \frac{\texttt{apply}(op, \overline{v_n}) = v}{\theta, C[op(\overline{v_n})], S \xrightarrow{\tau} \theta, C[v], S}$$

$$(Fun)\quad \frac{\texttt{current\_module}(S) = mod}{\theta, C[\texttt{fun } cl_1; \ldots; cl_n \texttt{ end}], S \xrightarrow{\tau} \theta, C[\langle \theta, mod, \overline{cl_n} \rangle], S}$$

$$(Call1)\quad \frac{\texttt{clauses}(mod, f, n, S) = \overline{cl_m} \quad \wedge \quad \texttt{match\_fun}(\overline{cl_m}, \overline{v_n}) = \langle \sigma, \overline{e} \rangle}{\theta, C[mod:f(\overline{v_n})], S \xrightarrow{\tau} \sigma, \overline{e}, \langle mod, \theta, C[\_] \rangle : S}$$

$$(Call2)\quad \frac{\texttt{match\_fun}(\overline{cl_n}, \overline{v_m}) = \langle \sigma, \overline{e} \rangle}{\theta, C[\langle \theta', mod, \overline{cl_n} \rangle (\overline{v_m})], S \xrightarrow{\tau} \theta'\sigma, \overline{e}, \langle mod, \theta, C[\_] \rangle : S}$$

$$(Return)\quad \frac{}{\sigma, v, \langle \theta, C[\_] \rangle : S \xrightarrow{\tau} \theta, C[v], S}$$

Figura 2.3: Semántica estándar: evaluación de expresiones secuenciales

la rama seleccionada. Al igual que antes, se almacena el contexto actual en la pila para evitar la aparición de expresiones ilegales si e_i es una secuencia. Si ninguna rama tiene una expresión que coincida se genera un error `case_clause`.

- La regla *Match* evalúa la ecuación usando la función auxiliar `match`, la cual devuelve la sustitución correspondiente en el caso de que el ajuste de patrones sea posible. En caso contrario, se genera un error `badmatch`.
- La regla *Op* evalúa operaciones aritméticas y relacionales usando la función auxiliar `apply`.
- La regla *Fun* evalúa una función anónima reduciéndola a una clausura.
- Las reglas *Call1* y *Call2* evalúan la aplicación de una función almacenando el contexto y entorno actual en la pila y entonces reduciendo el cuerpo de la función usando las funciones auxiliares `clauses`, la cual devuelve las ramas que componen la función "$mod{:}f/n$", y `match_fun`, que toma las ramas de la función y los argumentos de la llamada a la función. La regla *Return* recupera el contexto y el entorno de la pila una vez el cuerpo de la función ha sido reducido a un valor. En caso de que la función en cuestión

sea una BIF esta se evaluará por completo en un único paso y se devolverá el valor correspondiente.

Se puede deducir a partir de las reglas anteriores que la pila (S) contiene dos tipos de elementos:

- Cuando la ejecución cambia de contexto a un bloque `if` o `case`, se almacena únicamente la secuencia de expresiones que continúa fuera del bloque, es decir, $C[\_]$.
- Cuando se realiza una llamada a una función (anónima o no), se almacena una tripla con la forma $\langle mod, \theta, C[\_]\rangle$ donde mod es el módulo al que pertenece la función, θ es el entorno antes de entrar en la llamada y $C[\_]$ es la secuencia de expresiones que continúa fuera de la llamada a la función. La razón por la que se guarda el módulo de la función en la pila es para conocer qué funciones son accesibles desde el contexto actual; esto se consigue con la ayuda de la función auxiliar `clauses` que solo devuelve las ramas de la función indicada siempre que esta pertenezca al mismo módulo desde el que se realiza la llamada (o que esté exportada); en caso contrario, se genera un error `undef`.

2.2.2. Evaluación de expresiones concurrentes

En esta sección, se considera la semántica de las acciones concurrentes. Esencialmente, una aplicación en ejecución consiste en un número de procesos que interactúan entre ellos a través del envío y la recepción de mensajes. El envío de mensajes es asíncrono, mientras que la recepción bloquea el proceso si ningún mensaje (que coincida) ha llegado todavía. Los procesos se identifican de forma única mediante su *pid* (identificador de proceso).

En principio, se puede considerar que cada proceso tiene asociado un buzón o cola donde los mensajes entrantes se almacenan hasta que son consumidos por una expresión *receive*. Cuando un proceso envía un mensaje, este es almacenado en el buzón del proceso destinatario.[5] Entre el envío de un mensaje y su almacenaje en el buzón de un proceso, se dice que un mensaje está en la *red* (llamada *ether* en [8])

En este trabajo, al igual que en [4], la semántica abstrae las colas de los procesos. Por lo tanto, solo se considera una única estructura de datos, llamada *buzón global*, que representa tanto la red como el buzón de los procesos. Además, la semántica representa una sobreaproximación de la semántica real de Erlang ya que no se impone ninguna restricción sobre el orden de los mensajes. En Erlang, los mensajes entre dos procesos determinados deben llegar en el mismo orden en que se enviaron. Omitimos esta restricción por simplicidad (pero podría implementarse fácilmente, ver [6]) Sin embargo, eliminar esta restricción no es relevante en el modo de *replay*, ya que el depurador seguirá una traza de una ejecución real en algún entorno de Erlang y, por lo tanto, solo se pueden considerar las ejecuciones que respeten la restricción anterior.

Cabe destacar que esto contrasta con semánticas anteriores, por ejemplo, [3, 6, 8], donde tanto la red como los buzones de los procesos estaban modelados explícitamente.

A continuación se introducen las nociones de *proceso* y *sistema*.

[5] En este trabajo no se consideran los mensajes perdidos.

Definición 1 (proceso)**.** Un proceso está denotado por una configuración de la forma $\langle p, \theta, e, S\rangle$, donde p es el identificador del proceso, θ es un entorno (una substitución de variables por valores), e es una expresión a evaluar, y S es una pila.

A continuación, el *buzón global* se usa para almacenar los mensajes enviados hasta que se entregan (es decir, *consumidos* por un proceso usando una expresión *receive*):

Definición 2 (buzón global)**.** Un buzón global, Γ, se define como un multiconjunto de triplas con la forma $\langle pid\_destinatario, mensaje, etiqueta\rangle$. Dado un buzón global Γ, usamos $\Gamma \cup \langle p, v, \ell\rangle$ para denotar un nuevo buzón global que también incluye la tripla $\langle p, v, \ell\rangle$, donde "$\cup$" se usa como la unión de multiconjuntos.

Finalmente, un *sistema* se define como sigue:

Definición 3 (sistema)**.** Un sistema es un par $\Gamma; \Pi$, donde Γ es un buzón global y Π es una colección de procesos, denotada por $\langle p_1, \theta_1, e_1, S_1\rangle \mid \cdots \mid \langle p_n, \theta_n, e_n, S_n\rangle$; aquí "$\mid$" representa un operador asociativo y conmutativo. Normalmente un sistema se denota como $\Gamma; \langle p, \theta, e, S\rangle \mid \Pi$ para remarcar que $\langle p, \theta, e, S\rangle$ es una colección arbitraria de procesos (gracias al hecho de que "$\mid$" es asociativo y conmutativo).

Un *sistema inicial* tiene la forma $\{\}; \langle p, id, e, [\,]\rangle$, donde $\{\}$ es el buzón global vacío, p es un identificador de proceso, id es la sustitución identidad, e es una expresión (típicamente la aplicación de una función que inicia la ejecución), y $[\,]$ es una pila vacía.

Las reglas de transición para expresiones concurrentes se muestran en la Figura 2.4, mientras que las reglas de transición (etiquetadas) para sistemas se muestran en la Figura 2.5. Por el momento, se pueden ignorar las etiquetas de las flechas.

A continuación, se explica cómo se evalúan las expresiones concurrentes y los sistemas:

- Primero, un sistema en el cual el proceso seleccionado tiene una expresión secuencial (es decir, una expresión que se puede reducir usando las reglas de la Figura 2.3) se evalúa usando la regla *Seq* de la Figura 2.5 de la manera obvia.

- *Enviar un mensaje.* Las reglas *SendExp1* y *SendExp2* reducen las expresiones v_1`!`v_2 y `send`(v_1, v_2), respectivamente, es decir, el envío de un mensaje v_2 al proceso con el *pid* v_1. Sin embargo, también son necesarios ciertos efectos laterales: el mensaje v_2 debe ser recibido por el proceso v_1. Ya que esto no es observable de forma local, el paso es etiquetado con la etiqueta `send`(v_1, v_2) para que la regla *Send* se encargue de añadir la tripla $\langle p', v, \ell\rangle$ al buzón global Γ, donde p' es el *pid* del destinatario, v es el mensaje y ℓ es la etiqueta que identifica el mensaje. Los mensajes se etiquetan con etiquetas únicas para poder distinguirlos ya que, de otro modo, dos mensajes con el mismo valor serían indistinguibles.

- *Recibir un mensaje.* Al nivel de las expresiones, la regla *ReceiveExp* devuelve una variable fresca, κ, ya que las expresiones *receive* no se pueden

$$(SendExp1)\quad \frac{}{\theta, C[v_1\,!\,v_2], S \xrightarrow{\mathsf{send}(v_1,v_2)} \theta, C[v_2], S}$$

$$(SendExp2)\quad \frac{}{\theta, C[\mathtt{send}(v_1, v_2)], S \xrightarrow{\mathsf{send}(v_1,v_2)} \theta, C[v_2], S}$$

$$(ReceiveExp)\quad \frac{}{\theta, C[\mathtt{receive}\ cl_1; \ldots; cl_n\ \mathtt{end}], S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta, \kappa, C[\_]:S}$$

$$(SpawnExp1)\quad \frac{}{\theta, C[\mathtt{spawn}(mod, f, [\overline{v_n}])], S \xrightarrow{\mathsf{spawn}(\kappa,mod,f,\overline{v_n})} \theta, C[\kappa], S}$$

$$(SpawnExp2)\quad \frac{}{\theta, C[\mathtt{spawn}(\mathtt{fun}() \to e_1; \ldots; e_n\ \mathtt{end})], S \xrightarrow{\mathsf{spawn}(\kappa,\overline{e_n})} \theta, C[\kappa], S}$$

$$(SelfExp)\quad \frac{}{\theta, C[\mathtt{self}()], S \xrightarrow{\mathsf{self}(\kappa)} \theta, C[\kappa], S}$$

Figura 2.4: Semántica estándar: evaluación de expresiones con efectos laterales

$$(Seq)\quad \frac{\theta, e, S \xrightarrow{\tau} \theta', e', S'}{\Gamma; \langle p, \theta, e, S\rangle \mid \Pi \hookrightarrow_{p,\mathsf{seq}} \Gamma; \langle p, \theta', e', S'\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e, S \xrightarrow{\mathsf{send}(p',v)} \theta', e', S' \quad \wedge \quad \ell \text{ es un identificador nuevo}}{\Gamma; \langle p, \theta, e, S\rangle \mid \Pi \hookrightarrow_{p,\mathsf{send}(\ell)} \Gamma \cup \{\langle p', v, \ell\rangle\}; \langle p, \theta', e', S'\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e', S' \quad \wedge \quad \mathtt{match\_rec}(\overline{cl_n}\theta, v) = (\theta_i, e_i)}{\Gamma \cup \{\langle p, v, \ell\rangle\}; \langle p, \theta, e, S\rangle \mid \Pi \hookrightarrow_{p,\mathsf{rec}(\ell)} \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}, S'\rangle \mid \Pi}$$

$$(Spawn1)\quad \frac{\theta, e, S \xrightarrow{\mathsf{spawn}(\kappa,mod,f,\overline{v_n})} \theta', e', S' \quad \wedge \quad p' \text{ es un } pid \text{ nuevo}}{\Gamma; \langle p, \theta, e, S\rangle \mid \Pi \hookrightarrow_{p,\mathsf{spawn}(p')} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}, S'\rangle \mid \langle p', id, mod{:}f(\overline{v_n}), [\,]\rangle \mid \Pi}$$

$$(Spawn2)\quad \frac{\theta, e, S \xrightarrow{\mathsf{spawn}(\kappa,\overline{e_n})} \theta', e', S' \quad \wedge \quad p' \text{ es un } pid \text{ nuevo}}{\Gamma; \langle p, \theta, e, S\rangle \mid \Pi \hookrightarrow_{p,\mathsf{spawn}(p')} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}, S'\rangle \mid \langle p', id, \overline{e_n}, [\,]\rangle \mid \Pi}$$

$$(Self)\quad \frac{\theta, e, S \xrightarrow{\mathsf{self}(\kappa)} \theta', e', S'}{\Gamma; \langle p, \theta, e, S\rangle \mid \Pi \hookrightarrow_{p,\mathsf{self}} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p\}, S'\rangle \mid \Pi}$$

Figura 2.5: Semántica de trazas

reducir sin acceder al buzón global. κ puede verse como un "futuro" al cual se le asignará un valor en la siguiente capa de la semántica. Como antes, el paso se etiqueta con suficiente información para que la regla *Receive* pueda completar la reducción. Esta regla busca algún mensaje (en principio arbitrario) dirigido al proceso considerado en el buzón global, comprueba que empareje con alguna rama del la expresión *receive* usando la función auxiliar `match_rec`, y entonces procesa la evaluación de la rama en cuestión. La principal diferencia es que, ahora, κ tiene asignada la rama seleccionada y el mensaje es eliminado del buzón global.

- *Crear un proceso.* Para crear un proceso, se procede de forma análoga al caso anterior. Las reglas *SpawnExp1* y *SpawnExp2* etiquetan el paso con la información apropiada y las llamadas se reducen a una variable fresca κ. Entonces, las reglas *Spawn1* y *Spawn2* realizan el efecto lateral correspondiente (crear un nuevo proceso) y asignan a κ el *pid* del nuevo proceso.

- Finalmente, self() devuelve el *pid* del proceso actual usando las reglas *SelfExp* y *Self* de forma análoga a los casos anteriores.

A menudo se hace referencia a los pasos derivados de un sistema de reglas como *acciones* tomadas por el proceso elegido.

$$
\begin{aligned}
&\{\};\langle c,\_,\underline{\texttt{main()}},\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,\texttt{S} = \underline{\texttt{spawn(?MODULE, server, [])}},\ldots,\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,\texttt{P} = \underline{\texttt{spawn(?MODULE, proxy, [])}},\ldots,\_\rangle \mid \langle s,\_,\texttt{server()},\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,\underline{\texttt{client}(p,s)},\_\rangle \mid \langle s,\_,\texttt{server()},\_\rangle \mid \langle p,\_,\texttt{proxy()},\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,p\ \texttt{!}\ \{s,\{\underline{\texttt{self()}},40\}\}\ldots,\_\rangle \mid \langle s,\_,\texttt{server()},\_\rangle \mid \langle p,\_,\texttt{proxy()},\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,p\ \texttt{!}\ \{s,\{c,40\}\}\ldots,\_\rangle \mid \langle s,\_,\underline{\texttt{server()}},\_\rangle \mid \langle p,\_,\texttt{proxy()},\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,p\ \texttt{!}\ \{s,\{c,40\}\}\ldots,\_\rangle \mid \langle s,\_,\texttt{receive}\ldots,\_\rangle \mid \langle p,\_,\underline{\texttt{proxy()}},\_\rangle\\
\hookrightarrow\ &\{\};\langle c,\_,\underline{p\ \texttt{!}\ \{s,\{c,40\}\}}\ldots,\_\rangle \mid \langle s,\_,\texttt{receive}\ldots,\_\rangle \mid \langle p,\_,\texttt{receive}\ldots,\_\rangle\\
\hookrightarrow\ &\{\langle p,\{s,\{c,40\}\},\ell_1\rangle\};\langle c,\_,\underline{s\ \texttt{!}\ 2}\ldots,\_\rangle \mid \langle s,\_,\texttt{receive}\ldots,\_\rangle \mid \langle p,\_,\texttt{receive}\ldots,\_\rangle\\
\hookrightarrow\ &\{\langle p,\{s,\{c,40\}\},\ell_1\rangle,\langle s,2,\ell_2\rangle\};\langle c,\_,\texttt{receive}\ldots,\_\rangle \mid \langle s,\_,\texttt{receive}\ldots,\_\rangle \mid \langle p,\_,\underline{\texttt{receive}\ldots},\_\rangle\\
\hookrightarrow\ &\{\langle s,2,\ell_2\rangle\};\langle c,\_,\texttt{receive}\ldots,\_\rangle \mid \langle s,\_,\texttt{receive}\ldots,\_\rangle \mid \langle p,\_,\underline{s\ \texttt{!}\ \{c,40\}},\_\rangle\\
\hookrightarrow\ &\{\langle s,2,\ell_2\rangle,\langle s,\{c,40\},\ell_3\rangle\};\langle c,\_,\texttt{receive}\ldots,\_\rangle \mid \langle s,\_,\underline{\texttt{receive}\ldots},\_\rangle\\
\hookrightarrow\ &\{\langle s,\{c,40\},\ell_3\rangle\};\langle c,\_,\texttt{receive}\ldots,\_\rangle \mid \langle s,\_,\texttt{error},\_\rangle
\end{aligned}
$$

Figura 2.6: Derivación errónea con el cliente/servidor del Ejemplo 1

Ejemplo 2. Considerando el programa del Ejemplo 1 y el sistema inicial de la forma $\{\};\langle c, id, \texttt{main()}, []\rangle$, donde c es el *pid* del proceso. Una posible ejecución (errónea) de este sistema se muestra en la Figura 2.6 (la expresión seleccionada en cada paso está subrayada).[6] En este ejemplo, se ignoran las etiquetas de la relación $\hookrightarrow$. Además, se saltan los pasos que simplemente aplican variables y no se muestra el entorno (pero se sustituyen por sus valores por claridad), ni la pila.

[6] En términos generales, el problema viene del hecho de que los mensajes llegan al servidor en el orden incorrecto. Hay que tener en cuenta que esta derivación errónea es posible incluso considerando la política de Erlang sobre el orden de los mensajes, ya que siguen un camino diferente.

2.2.3. Concurrencia

Para definir una semántica reversible que sea causalmente consistente, necesitamos no solo una semántica como la que acabamos de presentar, sino también una noción de concurrencia (o, de manera equivalente, la noción opuesta de conflicto). Para ello, usamos las etiquetas de la semántica de trazas (vea la Figura 2.5). Estas etiquetas incluyen el *pid* p del proceso que realiza la transición, la regla utilizada para derivarla y, en algunos casos, alguna información adicional: una etiqueta de mensaje ℓ en las reglas *Send* y *Receive*, y el pid p' del proceso generado en las reglas *Spawn1* y *Spawn2*.

Antes de formalizar la noción de concurrencia, necesitamos introducir algo de notación y terminología. Dados los sistemas s_0, s_n, llamamos a $s_0 \hookrightarrow^* s_n$, que es una abreviatura de $s_0 \hookrightarrow_{p_1,r_1} \ldots \hookrightarrow_{p_n,r_n} s_n$, $n \geq 0$, una *derivación*. Las derivaciones de un paso simplemente se llaman *transiciones*. Usamos $d, d', d_1, \ldots$ para denotar derivaciones y $t, t', t_1, \ldots$ para transiciones.

Dada una derivación $d = (s_1 \hookrightarrow^* s_2)$, definimos $\mathsf{init}(d) = s_1$ y $\mathsf{final}(d) = s_2$. Dos derivaciones, d_1 y d_2, son *componibles* si $\mathsf{final}(d_1) = \mathsf{init}(d_2)$. En este caso, dejamos que $d_1; d_2$ denote su composición con $d_1; d_2 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$ si $d_1 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n)$ y $d_2 = (s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$. Dos derivaciones, d_1 y d_2, se dice que son *co-iniciales* si $\mathsf{init}(d_1) = \mathsf{init}(d_2)$, y *co-finales* si $\mathsf{final}(d_1) = \mathsf{final}(d_2)$.

Para simplificar, a continuación, consideramos derivaciones módulo renombramiento de variables vinculadas, es decir, decimos que las derivaciones d_1 y d_2 son iguales si son idénticas excepto (posiblemente) un renombramiento de las variables del entorno.

Bajo este supuesto, la semántica es *casi* determinista, es decir, las principales fuentes de indeterminismo son la selección de un proceso p (donde cualquier estrategia "fair" estaría bien, pero la dejamos sin especificar en este trabajo) y la selección del mensaje a ser recuperado en la regla *Receive* cuando más de un mensaje tiene como destinatario al proceso seleccionado p. También hay algo de indeterminismo en la elección del nuevo identificador ℓ para los mensajes en la regla *Send* y en la elección del pid p' del nuevo proceso en las reglas *Spawn1* y *Spawn2*. Sin embargo, los identificadores son solo un medio técnico para distinguir los mensajes, por lo que podemos considerarlos con seguridad hasta el cambio de nombre. También consideramos *pid*s módulo renombramiento, ya que en general su valor no es relevante, y esto simplifica la noción de concurrencia (de lo contrario, todas las aplicaciones de las reglas *Spawn1* y *Spawn2* estarían en conflicto debido a la selección del *pid*). Hay que tener en cuenta que cada proceso puede realizar como máximo una transición para cada etiqueta, es decir, $s \hookrightarrow_{p,r} s_1$ y $s \hookrightarrow_{p,r} s_2$ trivialmente implica $s_1 = s_2$.

Definición 4 (transiciones concurrentes)**.** Dadas dos transiciones co-iniciales diferentes, $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ y $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, decimos que están *en conflicto* si consideran el mismo proceso, es decir, $p_1 = p_2$, y las reglas aplicadas son *Receive*, es decir, $r_1 = \mathsf{rec}(\ell_1)$ y $r_2 = \mathsf{rec}(\ell_2)$ para algunos ℓ_1, ℓ_2 con $\ell_1 \neq \ell_2$. Dos transiciones co-iniciales diferentes son *concurrentes* si no están en conflicto.

2.2.4. Independencia

Ahora instanciamos en nuestro entorno la conocida relación *sucedió antes* [1], y la noción relacionada de transiciones *independientes*:[7]

Definición 5 (sucedió antes, independencia)**.** Dada una derivación d y dos transiciones $t_1 = (s_1 \hookrightarrow_{p_1,r_1} s_1')$ y $t_2 = (s_2 \hookrightarrow_{p_2,r_2} s_2')$ en d, decimos que t_1 sucedió antes de t_2, en símbolos $t_1 \rightsquigarrow t_2$, si se cumple una de las siguientes condiciones:

- consideran el mismo proceso, es decir, $p_1 = p_2$, y t_1 ocurre antes que t_2;
- t_1 crea un proceso p, es decir, $r_1 = \mathsf{spawn}(p)$, y t_2 se realiza en el proceso p, es decir, $p_2 = p$;
- t_1 envía un mensaje ℓ, es decir, $r_1 = \mathsf{send}(\ell)$, y t_2 recibe el mismo mensaje ℓ, es decir, $r_2 = \mathsf{rec}(\ell)$.

Además, si $t_1 \rightsquigarrow t_2$ y $t_2 \rightsquigarrow t_3$, entonces $t_1 \rightsquigarrow t_3$ (transitividad). Dos transiciones t_1 y t_2 son *independientes* si $t_1 \not\rightsquigarrow t_2$ y $t_2 \not\rightsquigarrow t_1$.

La relación sucedió antes da lugar a una relación de equivalencia que equipara todas las derivaciones que solo difieren en el intercambio de transiciones independientes. Formalmente,

Definición 6 (derivaciones causalmente equivalentes)**.** Sean d_1 y d_2 derivaciones con la semántica de trazas. Decimos que d_1 y d_2 son *causalmente equivalentes*, en símbolos $d_1 \approx d_2$, si d_1 se puede obtener a partir de d_2 mediante un número finito de intercambios de transiciones independientes consecutivas.

[7] Aquí, usamos el término *independiente* en lugar de *concurrente*, usado en [1], para evitar confusiones con la noción en la Definición 4, que es el significado típico del término concurrente en la literatura de la reversibilidad causalmente consistente.

Capítulo 3

Generando Trazas

En esta sección, se introduce la noción de *traza* de una ejecución. Básicamente, el objetivo es analizar en el depurador un comportamiento erróneo que ocurre en alguna ejecución de un programa. Para ello, es necesario extraer de una ejecución real suficiente información para reproducir dicha ejecución en un depurador. En realidad, no es necesario reproducir exactamente la misma ejecución, sino cualquiera que sea "causalmente equivalente". De este modo, el programador puede centrarse en las acciones de un proceso concreto, de forma que las acciones de otros procesos solo se realicen si es necesario (más formalmente, si existe una dependencia entre estas acciones). Tal y como se explica en la siguiente sección, esto asegura que los problemas en la ejecución se repetirán.

En una implementación práctica, uno debe instrumentar el programa para que su ejecución en el entorno real (por ejemplo, el entorno Erlang/OTP) produzca una colección de secuencias de trazas de eventos (una secuencia por proceso).

Se podría argumentar (como en, por ejemplo, [5]) que las trazas solo deben almacenar información sobre los eventos *receive*, ya que esta es la única acción indeterminista (una vez que se selecciona un proceso). Sin embargo, esto no es suficiente en este caso, donde:

- Es necesario trazar el envío de un mensaje ya que es donde los mensajes son etiquetados, y es necesario conocer su identificador (único) para ser capaz de relacionar el envío y la recepción de cada mensaje.
- Es necesario trazar la creación de nuevos procesos, ya que los *pid* generados son necesarios para relacionar una acción con el proceso donde se ha ejecutado.

Cabe mencionar que otros eventos no deterministas, como la entrada del usuario, también deberían ser trazados para poder reproducir correctamente las ejecuciones que los involucran. Uno puede lidiar con estos eventos instrumentando las primitivas correspondientes para trazar los valores de entrada y luego usar estos valores al reproducir la ejecución. Básicamente, pueden tratarse como la primitiva *receive*. Claramente, su tratamiento detallado sería necesario en una implementación de un depurador para Erlang completo. Sin embargo, se considera que este tema es ortogonal al enfoque del proyecto, y su manejo complicaría la semántica sin necesidad.

A continuación, se denota un conjunto (ordenado) de secuencias como $w = (r_1, \ldots, r_n)$, $n \geq 1$, donde [] representa una secuencia vacía. Dadas las secuencias w_1 y w_2, se define su concatenación como $w_1 + w_2$; cuando w_1 contiene un único elemento, es decir, $w_1 = (r)$, se escribe $r + w_2$ en lugar de $(r) + w_2$ por simplicidad.

Definición 7 (traza)**.** Una *traza* es una secuencia (finita) de eventos $(r_1, \ldots, r_n)$ donde cada r_i es uno de los siguientes: $\texttt{spawn}(p)$, $\texttt{send}(\ell)$ o $\texttt{rec}(\ell)$, donde p es un *pid* y ℓ un identificador de mensaje. Las trazas se representan por ω. Dada la derivación $d = (s_0 \hookrightarrow_{p_1,r_1} s_1 \hookrightarrow_{p_2,r_2} \ldots \hookrightarrow_{p_n,r_n} s_n)$, $n \geq 0$, la *traza de un proceso p en d*, simbolizada por $\mathcal{L}(d,p)$, se define inductivamente como sigue:

$$\mathcal{L}(d,p) = \begin{cases} [\,] & \text{si } n = 0 \ \vee \ p \text{ no aparece en } d \\ r_1 + \mathcal{L}(s_1 \hookrightarrow^* s_n, p) & \text{si } n > 0 \ \wedge \ p_1 = p \ \wedge \ r_1 \notin \{\mathsf{seq}, \mathsf{self}\} \\ \mathcal{L}(s_1 \hookrightarrow^* s_n, p) & \text{en otro caso} \end{cases}$$

La *traza de la derivación* d, $\mathcal{L}(d)$, se define como $\mathcal{L}(d) = \{(p, \mathcal{L}(d,p)) \mid p \text{ aparece en } d\}$. A veces se llama a $\mathcal{L}(d)$ la traza *global* de la derivación d para evitar confusiones con $\mathcal{L}(d,p)$. Trivialmente, $\mathcal{L}(d,p)$ se puede obtener a partir de $\mathcal{L}(d)$, es decir, $\mathcal{L}(d,p) = \omega$ si $(p,\omega) \in \mathcal{L}(d)$ y $\mathcal{L}(d,p) = [\,]$ en otro caso.

Ejemplo 3. Consideremos la derivación mostrada en el Ejemplo 2, aquí referida como d. La traza asociada sería la siguiente:

$$\begin{array}{rcl} \mathcal{L}(d,\mathrm{c}) & = & (\mathsf{spawn}(\mathrm{s}), \mathsf{spawn}(\mathrm{p}), \mathsf{send}(\ell_1), \mathsf{send}(\ell_2)) \\ \mathcal{L}(d,\mathrm{s}) & = & (\mathsf{rec}(\ell_2)) \\ \mathcal{L}(d,\mathrm{p}) & = & (\mathsf{rec}(\ell_1), \mathsf{send}(\ell_3)) \end{array}$$

Claramente, dada una derivación finita d, la traza $\mathcal{L}(d)$ asociada también es finita. Sin embargo, lo contrario no es cierto: podríamos tener una traza finita asociada a una derivación infinita (por ejemplo, aplicando un número infinito de veces la regla *Seq*).

A continuación, solo se consideran derivaciones finitas con la semántica de trazas. Esto es razonable en un contexto donde el programador quiere analizar en el depurador una ejecución finita (posiblemente incompleta) que mostró algún comportamiento erróneo.

Una propiedad esencial de la semántica es que las derivaciones causalmente equivalentes tienen la misma traza; es decir, la traza depende solo de la clase de equivalencia, no de la selección del representante dentro de la clase.

La implicación inversa, es decir, que las derivaciones (coiniciales) con la misma traza global son causalmente equivalentes, se cumple siempre que se establezca la siguiente convención sobre cuándo detener una derivación:

Definición 8 (derivación completamente trazada)**.** Una derivación d está *completamente trazada* si, para cada proceso p, su última transición $s_1 \hookrightarrow_{p,r} s_2$ en d es una transición *trazada*, es decir, $r \notin \{\mathsf{seq}, \mathsf{self}\}$.

Es necesario restringirse a las derivaciones completamente trazadas, ya que solo las transiciones trazadas contribuyen a las trazas. De lo contrario, dos derivaciones d_1 y d_2 podrían producir la misma traza, pero podrían diferir porque, por ejemplo, d_1 realiza más transiciones no trazadas que d_2.

Finalmente, se presenta un resultado clave de la semántica de trazas. Se establece que dos derivaciones son causalmente equivalentes si y solo si producen la misma traza.

Teorema 1. *Sean d_1, d_2 derivaciones coiniciales completamente trazadas.* $\mathcal{L}(d_1) = \mathcal{L}(d_2) \iff d_1 \approx d_2$.

La demostración de este teorema es similar a la presentada en [4] ya que las acciones concurrentes consideradas son las mismas en ambos casos.

Capítulo 4

Semántica reversible

En esta sección, primero presentamos una semántica instrumentada que es reversible, es decir, definimos un "Landauer embedding" apropiado para la semántica de la Figura 2.5. Luego, introducimos una semántica hacia atrás que procede en sentido contrario. Tanto la semántica hacia adelante (*forward*) como hacia atrás (*backward*) en esta sección son indeterministas. En el siguiente capítulo, se presentará otra capa por encima de estas reglas que se puede utilizar para guiar el proceso de depuración.

Ahora, los sistemas son triplas con la forma $\mathcal{W}; \Gamma; \Pi$, donde $\mathcal{W}$ es una *traza del sistema*, Γ es el buzón global y Π es la colección de procesos. Aquí, una traza de sistema se representa como una aplicación parcial de *pid*s a las trazas de los procesos (una traza vacía se denote mediante $[]$). El valor asociado al *pid* p en una traza de sistema $\mathcal{W}$ se denota mediante $\mathcal{W}[p]$, mientras que $\mathcal{W}[p \mapsto \omega]$ denota una traza de sistema $\mathcal{W}'$ con $\mathcal{W}'[p] = \omega$ y $\mathcal{W}'[p'] = \mathcal{W}[p']$ para todo $p' \neq p$. Se usa esta notación tanto como condición en una traza del sistema $\mathcal{W}$ o como una modificación de $\mathcal{W}$.

A continuación, las configuraciones del proceso se amplían con un nuevo componente: el *histórico* del proceso; es decir, las configuraciones de proceso tienen ahora la forma $\langle p, h, \theta, e, S\rangle$, donde p es el *pid* del proceso, h es un histórico del proceso, θ es un entorno, e es una expresión a evaluar y S es una pila. En este contexto, un histórico h registra los estados intermedios de un proceso usando términos encabezados por constructores seq, send, rec, spawn y self, y cuyos argumentos son la información requerida para deshacer (de forma determinista) el paso (siguiendo un "Landauer embedding" típico [2]).

$$(Seq)\quad \frac{\theta, e, S \xrightarrow{\tau} \theta', e', S'}{\mathcal{W}; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{seq},\{\mathsf{s}\}} \mathcal{W}; \Gamma; \langle p, \mathsf{seq}(\theta, e, S) + h, \theta', e', S'\rangle \mid \Pi}$$

$$(Send)\quad \frac{\theta, e, S \xrightarrow{\mathsf{send}(p',v)} \theta', e', S' \quad \wedge \quad \ell \text{ es un identificador nuevo}}{\mathcal{W}[p \mapsto [\,]]; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\uparrow\}} \mathcal{W}; \Gamma \cup \{\langle p', v, \ell\rangle\}; \langle p, \mathsf{send}(\theta, e, S, \langle p', v, \ell\rangle) + h, \theta', e', S'\rangle \mid \Pi}$$

$$\frac{\theta, e, S \xrightarrow{\mathsf{send}(p',v)} \theta', e', S'}{\mathcal{W}[p \mapsto \mathsf{send}(\ell) + \omega]; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\uparrow\}} \mathcal{W}[p \mapsto \omega]; \Gamma \cup \{\langle p', v, \ell\rangle\}; \langle p, \mathsf{send}(\theta, e, S, \langle p', v, \ell\rangle) + h, \theta', e', S'\rangle \mid \Pi}$$

$$(Receive)\quad \frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e', S' \quad \wedge \quad \texttt{match\_rec}(\overline{cl_n}\theta, v) = (\theta_i, e_i)}{\mathcal{W}[p \mapsto [\,]]; \Gamma \cup \{\langle p, v, \ell\rangle\}; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}} \mathcal{W}; \Gamma; \langle p, \mathsf{rec}(\theta, e, S, \langle p, v, \ell\rangle) + h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}, S'\rangle \mid \Pi}$$

$$\frac{\theta, e, S \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e', S' \quad \wedge \quad \texttt{match\_rec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\mathcal{W}[p \mapsto \mathsf{rec}(\ell) + \omega]; \Gamma \cup \{\langle p, v, \ell\rangle\}; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}} \mathcal{W}[p \mapsto \omega]; \Gamma; \langle p, \mathsf{rec}(\theta, e, S, \langle p, v, \ell\rangle) + h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}, S'\rangle \mid \Pi}$$

$$(Spawn1)\quad \frac{\theta, e, S \xrightarrow{\mathsf{spawn}(\kappa,mod,f,\overline{v_n})} \theta', e', S' \quad \wedge \quad p' \text{ es un } pid \text{ nuevo}}{\mathcal{W}[p \mapsto [\,]]; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \mathcal{W}; \Gamma; \langle p, \mathsf{spawn}(\theta, e, S, p') + h, \theta', e'\{\kappa \mapsto p'\}, S'\rangle \mid \langle p', [\,], id, mod{:}\, f(\overline{v_n}), [\,]\rangle \mid \Pi}$$

$$\frac{\theta, e, S \xrightarrow{\mathsf{spawn}(\kappa,mod,f,\overline{v_n})} \theta', e', S'}{\mathcal{W}[p \mapsto \mathsf{spawn}(p') + \omega]; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \mathcal{W}[p \mapsto \omega]; \Gamma; \langle p, \mathsf{spawn}(\theta, e, S, p') + h, \theta', e'\{\kappa \mapsto p'\}, S'\rangle \mid \langle p', [\,], id, mod{:}\, f(\overline{v_n}), [\,]\rangle \mid \Pi}$$

$$(Spawn2)\quad \frac{\theta, e, S \xrightarrow{\mathsf{spawn}(\kappa,\overline{e_n})} \theta', e', S' \quad \wedge \quad p' \text{ es un } pid \text{ nuevo}}{\mathcal{W}[p \mapsto [\,]]; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \mathcal{W}; \Gamma; \langle p, \mathsf{spawn}(\theta, e, S, p') + h, \theta', e'\{\kappa \mapsto p'\}, S'\rangle \mid \langle p', [\,], id, \overline{e_n}, [\,]\rangle \mid \Pi}$$

$$\frac{\theta, e, S \xrightarrow{\mathsf{spawn}(\kappa,\overline{e_n})} \theta', e', S'}{\mathcal{W}[p \mapsto \mathsf{spawn}(p') + \omega]; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \mathcal{W}[p \mapsto \omega]; \Gamma; \langle p, \mathsf{spawn}(\theta, e, S, p') + h, \theta', e'\{\kappa \mapsto p'\}, S'\rangle \mid \langle p', [\,], id, \overline{e_n}, [\,]\rangle \mid \Pi}$$

$$(Self)\quad \frac{\theta, e, S \xrightarrow{\mathsf{self}(\kappa)} \theta', e', S'}{\mathcal{W}; \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi \rightharpoonup_{p,\mathsf{self},\{\mathsf{s}\}} \mathcal{W}; \Gamma; \langle p, \mathsf{self}(\theta, e, S) + h, \theta', e'\{\kappa \mapsto p\}, S'\rangle \mid \Pi}$$

Figura 4.1: Semántica hacia adelante indeterminista

Las reglas de la semántica reversible hacia adelante (*forward*) se muestran en la Figura 4.1. Los subíndices de las flechas se pueden ignorar por ahora. Serán relevantes en la siguiente sección. A continuación, se explican brevemente las reglas de transición:

- La regla *Seq* procede como en la semántica original pero agrega un elemento $\mathsf{seq}(\theta, e, S)$ al histórico del proceso. Esta información es lo suficientemente trivial para deshacer el paso. De hecho, se puede optimizar esta representación siguiendo, por ejemplo, las ideas de [7].

- En cuanto al envío de un mensaje, se distinguen dos casos. Cuando la traza del proceso está vacía, se etiqueta el mensaje con un identificador nuevo; cuando la traza no está vacía, la etiqueta se obtiene de la traza. En ambos casos, se agrega un nuevo elemento de la forma $\mathsf{send}(\theta, e, S, \langle p', v, \ell\rangle)$ al histórico.

- Para la recepción de mensajes ocurre algo similar. Cuando la traza está vacía, simplemente se recibe el primer mensaje que empareje con alguna rama; cuando la traza no está vacía, se recibe el mensaje cuya etiqueta coincida con la de la traza. En ambos casos, se agrega un nuevo elemento de la forma $\mathsf{rec}(\theta, e, S, \langle p, v, \ell\rangle)$ al histórico.

- La creación de un proceso se consigue mediante las reglas *Spawn1* y *Spawn2*, y para ambas se distinguen dos casos. Cuando la traza está vacía, se genera un *pid* nuevo; cuando la traza no está vacía, se obtiene el *pid* de la traza. En ambos casos, se agrega un nuevo elemento de la forma $\mathsf{spawn}(\theta, e, S, p')$ al histórico.

- La regla *Self*, al igual que la primera regla, funciona igual que la semántica original pero agrega un elemento $\mathsf{self}(\theta, e, S)$ al histórico del proceso.

$$(\overline{Seq}) \quad \mathcal{W};\Gamma;\langle p,\mathsf{seq}(\theta,e,S)+h,\theta',e',S'\rangle \,|\, \Pi \;\leftharpoondown_{p,\mathsf{seq},\{\mathsf{s}\}\cup\mathcal{V}}\; \mathcal{W};\Gamma;\langle p,h,\theta,e,S\rangle \,|\, \Pi \quad \text{donde } \mathcal{V}=\mathcal{D}om(\theta')\backslash\mathcal{D}om(\theta)$$

$$(\overline{Send}) \quad \mathcal{W}[p\mapsto\omega];\Gamma\cup\{\langle p',v,\ell\rangle\};\langle p,\mathsf{send}(\theta,e,S,\langle p',v,\ell\rangle)+h,\theta',e',S'\rangle \,|\, \Pi \leftharpoondown_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\Uparrow\}} \mathcal{W}[p\mapsto\mathsf{send}(\ell)+\omega];\Gamma;\langle p,h,\theta,e,S\rangle \,|\, \Pi$$

$$(\overline{Receive}) \quad \mathcal{W}[p\mapsto\omega];\Gamma;\langle p,\mathsf{rec}(\theta,e,S,\langle p,v,\ell\rangle)+h,\theta',e',S'\rangle \,|\, \Pi \leftharpoondown_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}\cup\mathcal{V}} \mathcal{W}[p\mapsto\mathsf{rec}(\ell)+\omega];\Gamma\cup\{\langle p,v,\ell\rangle\};\langle p,h,\theta,e,S\rangle \,|\, \Pi \quad \text{donde } \mathcal{V}=\mathcal{D}om(\theta')\backslash\mathcal{D}om(\theta)$$

$$(\overline{Spawn}) \quad \mathcal{W}[p\mapsto\omega];\Gamma;\langle p,\mathsf{spawn}(\theta,e,S,p')+h,\theta',e',S'\rangle \,|\, \langle p',[\,],id,e'',[\,]\rangle \,|\, \Pi \leftharpoondown_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \mathcal{W}[p\mapsto\mathsf{spawn}(p')+\omega];\Gamma;\langle p,h,\theta,e,S\rangle \,|\, \Pi$$

$$(\overline{Self}) \quad \mathcal{W};\Gamma;\langle p,\mathsf{self}(\theta,e,S)+h,\theta',e',S'\rangle \,|\, \Pi \leftharpoondown_{p,\mathsf{self},\{\mathsf{s}\}} \mathcal{W};\Gamma;\langle p,h,\theta,e,S\rangle \,|\, \Pi$$

Figura 4.2: Semántica hacia atrás indeterminista

Las reglas de la semántica hacia atrás (*backward*) se muestran en la Figura 4.2. Aquí únicamente tenemos cinco reglas que son análogas a las de la Figura 4.1, y se aplican a sistemas con tazas no vaciás. Las reglas *Spawn1* y *Spawn2* tienen como análoga la misma regla, $\overline{Spawn}$. Ya que toda la información necesaria para volver al estado anterior esta almacenada en el histórico, no es necesario crear dos reglas diferentes.

4.1. Utilidad para depurar

En esta sección, mostramos que nuestra semántica reversible es útil como base para diseñar una herramienta de depuración. En particular, demostramos que ocurre un comportamiento (erróneo) en la derivación trazada si y solo si la derivación de la repetición también exhibe el mismo comportamiento *erróneo*, por lo tanto, la repetición es correcta y completa. En particular, la corrección y la integridad no dependen de la planificación (*scheduling*).

Para formalizar tal resultado, necesitamos especificar la noción de comportamiento erróneo que nos interesa. Para nosotros, un mal comportamiento es un sistema incorrecto, pero dado que el sistema está posiblemente distribuido, nos concentramos en los malos comportamientos visibles desde un punto de vista de un observador "local". Puesto que nuestros sistemas están compuestos por procesos y mensajes en el buzón global, consideramos que un mal comportamiento (local) es un mensaje incorrecto en el buzón global o un proceso con una configuración incorrecta.

Definición 9. Sea $d = (s_1 \hookrightarrow^* s_2)$ una derivación con la semántica de trazas, donde

$$s_1 = \Gamma; \langle p_1, \theta_1, e_1, S_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n, S_n\rangle$$

El sistema correspondiente a s_1 en la semántica reversible se define de la siguiente manera:

$$addLog(\mathcal{L}(d), s_1) = \mathcal{L}(d); \Gamma; \langle p_1, [\,], \theta_1, e_1, S_1\rangle \mid \ldots \mid \langle p_n, [\,], \theta_n, e_n, S_n\rangle$$

A la inversa, dado un sistema

$$s = \mathcal{W}; \Gamma; \langle p_1, h_1, \theta_1, e_1, S_1\rangle \mid \ldots \mid \langle p_n, h_n, \theta_n, e_n, S_n\rangle$$

en la semántica reversible, $delLog(s)$ se define como el sistema obtenido de s al eliminar tanto los registros como los históricos, es decir,

$$delLog(s) = \Gamma; \langle p_1, \theta_1, e_1, S_1\rangle \mid \ldots \mid \langle p_n, \theta_n, e_n, S_n\rangle$$

Esta definición se extiende a derivaciones de la manera obvia: dada una derivación d de la forma $s_1 \hookrightarrow \ldots \hookrightarrow s_n$, decimos que $delLog(d)$ es $delLog(s_1) \hookrightarrow \ldots \hookrightarrow delLog(s_n)$.

Esencialmente, la función *addLog* equipa al sistema con una traza dada y a cada proceso con un historial vacío.

Teorema 2 (Correctitud y completitud)**.** *Sea d una derivación completamente trazada bajo la semántica de trazas. Sea d' cualquier derivación completamente trazada bajo la semántica hacia adelante indeterminista tal que* $\mathsf{init}(d') = addLog(\mathcal{L}(d), \mathsf{init}(d))$. *Entonces:*

1. *hay un sistema* $\Gamma; \Pi$ *en* d *con una configuración* $\langle p, \theta, e, S\rangle$ *en* Π *si y solo si hay un sistema* $\mathcal{W}'; \Gamma'; \Pi'$ *en* d' *con una configuración* $\langle p, \theta, e, S\rangle$ *en* $delLog(\mathcal{W}'; \Gamma'; \Pi')$*;*

2. *hay un sistema* $\mathcal{W}; \Gamma; \Pi$ *en* d *con un mensaje* $\langle p', v, \ell\rangle$ *en* Γ *si y solo si hay un sistema* $\mathcal{W}'; \Gamma'; \Pi'$ *en* d' *con un mensaje* $\langle p', v, \ell\rangle$ *en* Γ'*.*

De nuevo, no se muestra la demostración del teorema anterior ya que esta es similar a la demostración mostrada en [4].

Capítulo 5

Semántica determinista

En esta sección, presentamos una versión determinista de la semántica reversible. La idea clave es que esta semántica está guiada por las peticiones del usuario como, por ejemplo, "avanza hasta la creación del proceso p" o "retrocede hasta el paso anterior al envío del mensaje ℓ".

Aquí consideramos que, dado un sistema s, queremos iniciar una repetición (resp. reversión) hasta que se realice (resp. deshaga) una acción particular ψ en un proceso p dado. Se denota dicha petición con la siguiente notación: $[\![s]\!]_{\Phi}$, donde s es un sistema y Φ es una secuencia de peticiones que se puede ver como una pila donde el primer elemento es la petición más reciente. Las peticiones se formalizan como un flujo estático que se proporciona al cálculo pero, en la práctica, las peticiones se proporcionan por el usuario de forma interactiva. En este documento, consideramos las siguientes peticiones de reversión/repetición:

- $\{p, \mathsf{s}\}$: un paso hacia atrás/adelante en el proceso p.[1]
- $\{p, \ell_{\Uparrow}\}$: una derivación hacia atrás/adelante del proceso p hasta el envío del mensaje etiquetado con ℓ.
- $\{p, \ell_{\Downarrow}\}$: una derivación hacia atrás/adelante del proceso p hasta la recepción del mensaje etiquetado con ℓ.
- $\{p, \mathsf{sp}_{p'}\}$: una derivación hacia atrás/adelante del proceso p hasta la creación del proceso con el *pid* p'.
- $\{p, \mathsf{sp}\}$: una derivación hacia atrás del proceso p hasta el punto inmediatamente anterior a su creación.
- $\{p, X\}$: una derivación hacia atrás del proceso p hasta la introducción de la variable X.

Cuando la petición puede ser de repetición o reversión, usamos una flecha para indicar la dirección. Por ejemplo, $\{p, \overrightarrow{\mathsf{s}}\}$ requiere un paso hacia adelante, mientras que $\{p, \overleftarrow{\mathsf{s}}\}$ requiere un paso hacia atrás. En particular, $\{p, \overleftarrow{\mathsf{sp}}\}$ y $\{p, \overleftarrow{X}\}$ tienen solo una versión ya que siempre requieren un cálculo hacia atrás.

Además, a diferencia de enfoques anteriores, agregamos una nueva petición, $\{p, \overrightarrow{\ell_{\Downarrow\Downarrow}}\}$, que fuerza la recepción del mensaje etiquetado con ℓ en lugar del

[1] La extensión a n pasos es sencilla. Se omite por simplicidad.

mensaje que se debe recibir de acuerdo con la traza actual. Por lo general, se llega a un sistema donde la traza requiere la recepción del mensaje ℓ' y es posible un paso de recepción, pero el usuario prefiere explorar otra alternativa y recibir el mensaje ℓ en su lugar. En este caso, la traza restante, así como las acciones de otras trazas que dependen de la traza borrada, también se eliminan mediante la función auxiliar *rdep*.

No se incluyen las creaciones de variables como objetivos para las peticiones de repetición, ya que los nombres de las variables no se conocen antes de su creación (las creaciones de variables no se registran). Las peticiones anteriores se *satisfacen* cuando se realiza una transición indeterminista correspondiente. Aquí es donde entra en juego el tercer elemento que etiqueta las relaciones de la semántica reversible en las Figuras 4.1 y 4.2. Este tercer elemento es un conjunto con las peticiones que se satisfacen en el paso correspondiente.

$$\frac{}{[\![\_;\Gamma;\Pi]\!]_{newlog(\mathcal{W})+\Psi} \rightsquigarrow [\![\mathcal{W};\Gamma;\Pi]\!]_{\Psi}} \tag{1}$$

$$\frac{}{[\![\_;\Gamma;\Pi]\!]_{exitlog+\Psi} \rightsquigarrow [\![[\,];\Gamma;\Pi]\!]_{\Psi}} \tag{2}$$

$$\frac{\mathcal{W};\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \mathcal{W}';\Gamma';\Pi' \quad\wedge\quad \psi \in \Psi'}{[\![\mathcal{W};\Gamma;\Pi]\!]_{\{p,\overrightarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W}';\Gamma';\Pi']\!]_{\Psi}} \tag{3}$$

$$\frac{\mathcal{W};\Gamma;\Pi \rightharpoonup_{p,r,\Psi'} \mathcal{W}';\Gamma';\Pi' \quad\wedge\quad \psi \notin \Psi'}{[\![\mathcal{W};\Gamma;\Pi]\!]_{\{p,\overrightarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W}';\Gamma';\Pi']\!]_{\{p,\overrightarrow{\psi}\}+\Psi}} \tag{4}$$

$$\frac{\mathcal{W}[p \mapsto \mathsf{rec}(\ell)+\omega];\Gamma;\Pi \not\rightharpoonup_{p,r,\Psi'} \quad\wedge\quad sender(\mathcal{W},\ell) = p'}{[\![\mathcal{W}[p \mapsto \mathsf{rec}(\ell)+\omega];\Gamma;\Pi]\!]_{\{p,\overrightarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W}[p \mapsto \mathsf{rec}(\ell)+\omega];\Gamma;\Pi]\!]_{(\{p',\overrightarrow{\ell_{\Uparrow}}\},\{p,\overrightarrow{\psi}\})+\Psi}} \tag{5}$$

$$\frac{\mathcal{W}[p \mapsto \mathsf{rec}(\ell')+\omega];\Gamma;\Pi \not\rightharpoonup_{p,r,\Psi'} \quad\wedge\quad sender(\mathcal{W},\ell) = p' \quad\wedge\quad \ell' \neq \ell}{[\![\mathcal{W}[p \mapsto \mathsf{rec}(\ell')+\omega];\Gamma;\Pi]\!]_{\{p,\overrightarrow{\ell_{\Downarrow\Downarrow}}\}+\Psi} \rightsquigarrow [\![\mathcal{W}[p \mapsto \mathsf{rec}(\ell')+\omega];\Gamma;\Pi]\!]_{(\{p',\overrightarrow{\ell_{\Uparrow}}\},\{p,\overrightarrow{\ell_{\Downarrow\Downarrow}}\})+\Psi}} \tag{6}$$

$$\frac{\mathcal{W}[p \mapsto \mathsf{rec}(\ell')+\omega];\Gamma;\Pi \rightharpoonup_{p,\mathsf{rec}(\ell),\Psi'} \mathcal{W}';\Gamma';\Pi' \quad\wedge\quad \ell \neq \ell'}{[\![\mathcal{W}[p \mapsto \mathsf{rec}(\ell')+\omega];\Gamma;\Pi]\!]_{\{p,\overrightarrow{\ell_{\Downarrow\Downarrow}}\}+\Psi} \rightsquigarrow [\![rdep(p,\mathcal{W}');\Gamma';\Pi']\!]_{\Psi}} \tag{7}$$

$$\frac{\nexists p \text{ in } \Pi \quad\wedge\quad parent(\mathcal{W},p) = p'}{[\![\mathcal{W},\Gamma;\Pi]\!]_{\{p,\overrightarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W},\Gamma;\Pi]\!]_{(\{p',\overrightarrow{\mathsf{sp}_p}\},\{p,\overrightarrow{\psi}\})+\Psi}} \tag{8}$$

Figura 5.1: Semántica de repetición determinista

A continuación, se explican las reglas de la semántica de repetición determinista de la Figura 5.1. Aquí, asumimos que el cálculo siempre comienza con una sola petición. Tenemos las siguientes posibilidades:

- Si se recibe una petición $newlog(\mathcal{W})$ se ignorará la traza actual, y se cargará la nueva traza $\mathcal{W}$ de modo que se podrá repetir pasos (1).
- Si se recibe una petición *exitlog* se vaciará la traza actual, de modo que ya no se podrán repetir más pasos ya que la traza ahora está vacía (2).
- Si el proceso deseado p puede realizar un paso que satisface la petición ψ en la parte superior de la pila, lo realiza y se elimina la petición de la pila de peticiones (3).

- Si el proceso deseado p puede realizar un paso, pero el paso no satisface la petición ψ, actualizamos el sistema pero mantenemos la petición en la pila (4).

- Si un paso en el proceso deseado p no es posible, seguimos las dependencias y agregamos una nueva petición en la parte superior de la pila. Tenemos dos reglas: una para agregar una petición a un proceso para enviar un mensaje que queremos recibir (5) y otra para generar el proceso que queremos repetir si no existe (8). Aquí, usamos las funciones auxiliares *sender* y *parent* para identificar, respectivamente, el remitente de un mensaje y el padre de un proceso.

- Si el usuario quiere forzar la recepción de un mensaje distinto del que debería de acuerdo a la traza, seguimos las dependencias y agregamos una nueva petición en la parte superior de la pila. Tenemos tres reglas: una para agregar una petición a un proceso para enviar un mensaje que queremos recibir (6), la misma que antes para generar el proceso que queremos repetir si no existe (8) y una otra vara invalida las trazas correspondientes mediante la función auxiliar *rdep*.

Ambas funciones *sender* y *parent* se pueden calcular fácilmente a partir de las trazas en $\mathcal{L}(d)$.[2] La función $rdep(p, \mathcal{W})$ se encarga de borrar la traza del proceso p y a continuación, realiza las siguientes acciones dependiendo de las acciones que hubiese en la traza: para cada $spawn(p')$, borra la traza del proceso p'; y para cada $send(\ell)$, debe buscar la acción $rec(\ell)$ correspondiente y entonces borrar las trazas desde esa acción en adelante.[3]

[2] La implementación de las funciones *sender* y *parent* requiere cierto cuidado: sin información adicional, necesitan explorar la traza completa. Sin embargo, la representación explícita de las funciones se puede calcular fácilmente durante el trazado o mediante una única pasada de preprocesamiento en la traza. Por lo tanto, existe un compromiso entre el tiempo de repetición, el tamaño de la traza y el tiempo de otras fases.

[3] Hay que tener en cuenta que la función *rdep* debe definirse de forma recursiva, ya que eliminar las trazas de otros procesos puede desencadenar más eliminaciones.

$$\frac{\mathcal{W};\Gamma;\Pi \leftharpoondown_{p,r,\Psi'} \mathcal{W}';\Gamma';\Pi' \quad \wedge \quad \psi \in \Psi'}{[\![\mathcal{W};\Gamma;\Pi]\!]_{\{p,\overleftarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W}';\Gamma';\Pi']\!]_{\Psi}} \tag{1}$$

$$\frac{\mathcal{W};\Gamma;\Pi \leftharpoondown_{p,r,\Psi'} \mathcal{W}';\Gamma';\Pi' \quad \wedge \quad \psi \notin \Psi'}{[\![\mathcal{W};\Gamma;\Pi]\!]_{\{p,\overleftarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W}';\Gamma';\Pi']\!]_{\{p,\overleftarrow{\psi}\}+\Psi}} \tag{2}$$

$$\frac{\mathcal{W};\Gamma;\langle p, \mathsf{send}(\theta,e,S,\langle p,v,\ell\rangle)+h,\theta',e',S'\rangle \mid \Pi \not\leftharpoondown_{p,r,\Psi'}}{[\![\mathcal{W};\Gamma;\langle p, \mathsf{send}(\theta,e,S,\langle p,v,\ell\rangle)+h,\theta',e',S'\rangle \mid \Pi]\!]_{\{p,\overleftarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W};\Gamma;\langle p, \mathsf{send}(\theta,e,S,\langle p,v,\ell\rangle)+h,\theta',e',S'\rangle \mid \Pi]\!]_{(\{p',\overleftarrow{\ell_{\Downarrow}}\},\{p,\overleftarrow{\psi}\})+\Psi}} \tag{3}$$

$$\frac{\mathcal{W};\Gamma;\langle p, \mathsf{spawn}(\theta,e,S,p')+h,\theta',e',S'\rangle \mid \Pi \not\leftharpoondown_{p,r,\Psi'}}{[\![\mathcal{W};\Gamma;\langle p, \mathsf{spawn}(\theta,e,S,p')+h,\theta',e',S'\rangle \mid \Pi]\!]_{\{p,\overleftarrow{\psi}\}+\Psi} \rightsquigarrow [\![\mathcal{W};\Gamma;\langle p, \mathsf{spawn}(\theta,e,S,p')+h,\theta',e',S'\rangle \mid \Pi]\!]_{(\{p',\overleftarrow{\mathsf{sp}}\},\{p,\overleftarrow{\psi}\})+\Psi}} \tag{4}$$

$$\frac{}{[\![\mathcal{W};\Gamma;\langle p, [\,],\theta',e',S'\rangle \mid \Pi]\!]_{\{p,\overleftarrow{\mathsf{sp}}\}+\Psi} \rightsquigarrow [\![\mathcal{W};\Gamma;\langle p, [\,],\theta',e',S'\rangle \mid \Pi]\!]_{\Psi}} \tag{5}$$

Figura 5.2: Semántica de reversión determinista

Las reglas de la semántica de reversión de la Figura 5.2 son muy similares a las de la semántica de repetición que acabamos de ver. A continuación, se explican las diferencias:

- Las dos primeras reglas (1 y 2) son equivalentes a las tercera (3) y cuarta (4) de la semántica de repetición.
- Cuando no es posible realizar un paso en el proceso deseado p, al igual que antes, seguimos las dependencias y agregamos una nueva petición en la parte superior de la pila. En este caso tenemos tres reglas: una para agregar una petición para deshacer la recepción de un mensaje cuyo envío queremos deshacer (3), otra para deshacer las acciones de un proceso dado cuya creación queremos deshacer (4), y una última para comprobar que un proceso haya alcanzado su estado inicial (con un historial vacío) y que la petición $\{p, \mathsf{sp}\}$ se pueda eliminar (5). En este último caso, el proceso p se eliminará del sistema cuando una petición de la forma $\{p', sp_p\}$ esté en la parte superior de la pila.

La relación $\rightsquigarrow$ puede verse como una versión determinista de la semántica reversible indeterminista en el sentido de que cada derivación de la semántica determinista corresponde a una derivación de la indeterminista, mientras que lo contrario no es cierto generalmente.

Capítulo 6

CauDEr 2.0

En este capítulo se aborda el funcionamiento de la nueva versión del depurador: CauDEr 2.0[1], comparándolo con la versión anterior y explorando posibles mejoras en un futuro.

6.1. Desarrollo

Al igual que la versión anterior, esta nueva versión de CauDEr está escrita completamente en Erlang, tanto la lógica como la interfaz de usuario, la cual se explica en detalle en la siguiente sección. Para la interfaz de usuario se hace uso de la "aplicación" `wx`[2] que viene integrada por defecto en la distribución de Erlang/OTP, y la cual consiste en unos *bindings* de Erlang para la biblioteca wxWidgets, escrita en C++.

La versión original de CauDEr se desarrolló para trabajar con Core Erlang. Core Erlang es una representación intermedia de Erlang, destinada a ubicarse en un nivel entre el código fuente y el código intermedio que normalmente se encuentra en los compiladores. La sintaxis aunque es muy reducida en comparación con el lenguaje Erlang, puede llegar a ser complicada de entender si uno se encuentra con ciertas construcciones. En [4] se puede observar que el subconjunto de Core Erlang usado en la versión anterior de CauDEr apenas comprende siete reglas de sintaxis.

Por otra parte, la nueva versión de CauDEr hace uso del *abstract format*[3] que básicamente son árboles de sintaxis, los cuales permiten trabajar con el código fuente sin realizar ningún tipo de simplificación, como sí ocurría al trabajar con Core Erlang; por ejemplo, en la Figura 2.1 se puede observar que el subconjunto de Erlang usado en esta nueva versión comprende veinte reglas, en comparación con las siete de la versión anterior. Además, ya que se trata de una representación equivalente es posible extraer información relevante como el número de línea en el código fuente de una expresión en *abstract format*, lo que permite saber en que línea se encuentra la ejecución (algo que no era posible en la versión anterior). Otro acontecimiento que ha motivado el abandono

[1] CauDEr 2.0 está disponible en: `https://github.com/mistupv/cauder-v2`

[2] Documentación sobre `wx` disponible en: `https://erlang.org/doc/apps/wx/chapter.html`

[3] Definición del *abstract format* disponible en: `https://erlang.org/doc/apps/erts/absform.html`

de Core Erlang ha sido un *breaking change* en la versión 23 de Erlang/OTP, donde las expresiones *receive* han sido remplazadas por operaciones primitivas (*primops*), de modo que CauDEr no puede trabajar con esta nueva versión, y de actualizarse dificultaría al usuario el uso del depurador ya que se introducirían construcciones más complejas.[4] Es por ello que ha sido necesario reescribir prácticamente por completo la parte de la lógica de la depuración y, en menor medida la interfaz.

La distribución de Erlang/OTP al igual que proporciona herramientas para trabajar con Core Erlang, también incluye otras herramientas para trabajar con el formato abstracto de Erlang. Esto ha facilitado mucho el desarrollo inicial de la nueva versión, aunque a medida que ha avanzado el desarrollo y se ha ido añadiendo soporte para construcciones más complejas del lenguaje, como pueden ser las funciones anónimas, ha sido necesario crear una representación personalizada a partir del formato abstracto para poder trabajar de forma correcta con estas construcciones, algo similar a lo que hace el propio depurador de Erlang. El formato abstracto original solo contempla su representación en forma de código fuente pero no en tiempo de ejecución; por ejemplo, una función anónima en el código fuente puede tener una representación similar a `fun(X) → X * 2 end`, pero en tiempo de ejecución su representación pasa a ser algo similar a `#Fun<erl_eval.44.97283095>`.

6.2. Interfaz

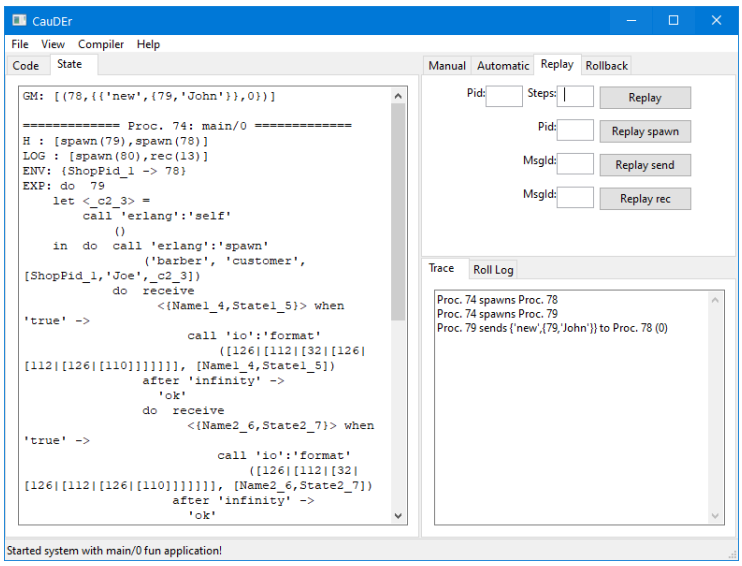

Figura 6.1: Ventana principal de la versión anterior de CauDEr

Como se comentaba en la sección anterior, la interfaz también se ha actualizado para mostrar tanto el código del programa que se está depurando como la información sobre el estado de la sesión de depuración de una forma más

[4] Entrada en el *Issue Tracker* de Erlang, sobre el cambio de las expresiones *receive*: `https://bugs.erlang.org/browse/ERL-1277`

intuitiva y fácil de entender para el usuario, ya que en la versión anterior el código no tenia ningún tipo de resaltado de sintaxis y toda la información sobre el sistema y los procesos se mostraba en el mismo cuadro de texto plano, sin apenas ningún formato, tal y como se observa en la Figura 6.1. En esta figura también se puede observar la representación de un programa en Core Erlang el cual, aunque se conozca, cuesta de entender a medida que crece el tamaño del código.

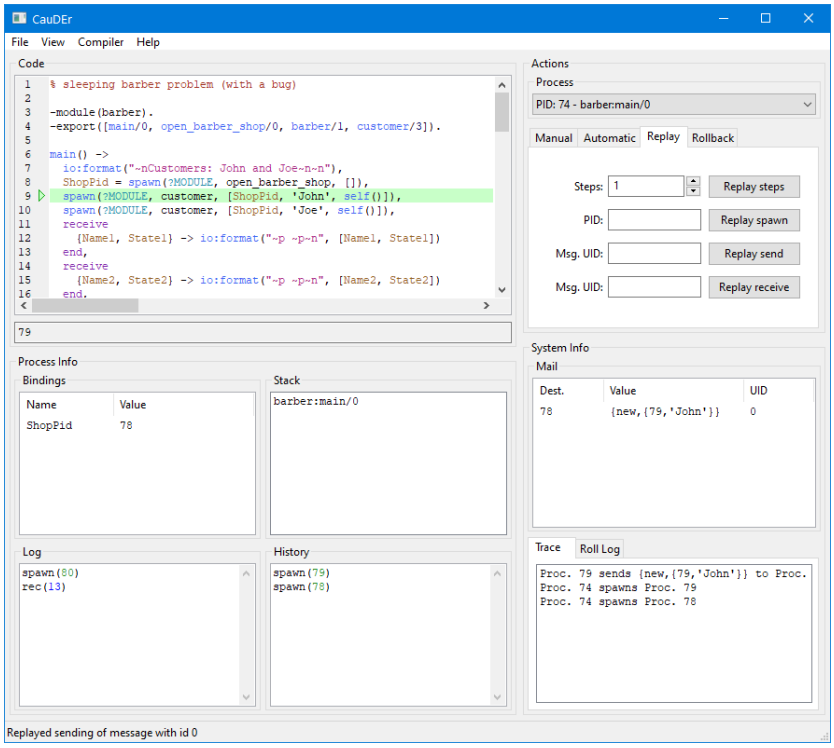

Figura 6.2: Ventana principal de la nueva versión de CauDEr

Al ejecutar la nueva versión aparece una ventana como la que se muestra en la Figura 6.2, desde la cual se puede iniciar y controlar una sesión de depuración. Esta ventana principal está dividida en cuatro áreas, las cuales se explican a continuación.

6.2.1. Código

El área de código se encuentra en la esquina superior izquierda y consiste en un cuadro de texto. En este cuadro de texto se muestra el código del programa que se haya cargado. Este cuadro usa un tipo de letra Courier y posee resaltado de sintaxis. Una vez se haya iniciado una sesión de depuración el contenido de este área cambiará a lo largo del tiempo para mostrar el punto en el que se encuentre la ejecución del proceso que esté seleccionado.

Además, en la parte inferior del area de código hay un cuadro de texto que muestra cómo se evalúa paso a paso la expresión actual (marcada en verde en el cuadro anterior).

6.2.2. Información del proceso

La información relativa al proceso actualmente seleccionado se encuentra en la esquina inferior izquierda, bajo el título *Process Info*, y consta de cuatro elementos:

- *Bindings* muestra las variables declaradas en el ámbito actual, relativo a la línea actual (marcada en verde).
- *Stack* es la pila de llamadas a función, y por temas de implementación también muestra los "bloques" en los que se encuentra; es decir, si entramos en un `if`, `case` o `receive` también se mostrará en este cuadro.
- *Log*, en caso de que exista una traza a seguir de una ejecución anterior, esta se mostrará aquí.
- *History*, aquí aparece cada paso que se realiza; se podría decir que es la inversa del *Log*.

6.2.3. Acciones

El área de acciones se encuentra en la esquina superior derecha, bajo el titulo *Actions*, y consta de dos partes:

- Un selector de procesos en la parte superior, bajo el título *Process*, donde se elige el proceso sobre el cual se aplicarán las acciones.
- Un cuadro con las diferentes acciones agrupadas en pestañas dependiendo de su categoría. Estas categorías junto con las acciones que contienen se explican a continuación.

Manual

En esta pestaña tenemos botones con los títulos *Backward* y *Forward*, y cuya etiqueta cambiará dependiendo del tipo de regla que se vaya a aplicar para reducir la expresión actual. Los posibles valores de las etiquetas son los nombres de las reglas de la Figura 2.5 escritos en minúscula. La única diferencia es que las reglas *Spawn1* y *Spawn2* están representadas por la misma etiqueta, *spawn*.

Estos botones sirven para avanzar o retroceder respectivamente un paso, aplicando la regla que en ese momento aparezca en el botón, en la expresión actual (marcada en verde) del proceso actualmente seleccionado en el selector mencionado anteriormente. Por este motivo es posible que sea necesario pulsar varias veces el mismo botón para cambiar de expresión.

Se ha considerado en una futura versión implementar dos nuevas acciones: *Step Over* que permita evaluar por completo la expresión actual y avanzar a la siguiente expresión; y *Step Into* que permita entrar dentro de la definición de una función y continuar desde ahí la ejecución.

Automatic

Aquí de nuevo tenemos dos botones, uno para avanzar y otro para retroceder, con la adición de un cuadro de texto donde podemos indicar la cantidad de pasos de reducción a realizar. La única diferencia es que en este caso los pasos

se realizan de forma aleatoria entre todos los procesos disponibles, en vez de en el que esté seleccionado, siempre que se pueda avanzar (por ejemplo, cuando el proceso no esté bloqueado esperando la recepción de un mensaje).

Replay

En esta pestaña se encuentran los controles para reproducir los eventos de una traza que haya sido cargada previamente, de modo que esta pestaña solo estará activa si se ha cargado una traza. Cada acción de esta pestaña consume elementos de la traza (*Log*) y añade elementos al histórico (*History*). Las acciones disponibles son las que siguen:

- *Replay steps*, reproduce el número indicado de pasos, de acuerdo a las reglas de la Figura 5.1, sobre el proceso actualmente seleccionado y con el orden establecido en la traza mostrada en cuadro *Log*.
- *Replay spawn*, reproduce la creación del proceso con el *pid* indicado, reproduciendo antes y en orden todos los pasos de los cuales dependa este, y solo si existe algún proceso con dicho *pid* en la traza.
- *Replay send*, realiza una acción similar solo que en este caso reproduce el envío del mensaje cuya etiqueta sea la indicada, y de nuevo solo si existe algún mensaje con dicha etiqueta en la traza.
- *Replay receive*, hace lo mismo que la acción anterior solo que en este caso además de reproducir la recepción del mensaje con la etiqueta indicada, también reproduce la creación del proceso que envía el mensaje con dicha etiqueta y su envío, ya que si no se envía el mensaje no se puede recibir. Y al igual que con las acciones anteriores solo se realiza si existe algún mensaje con la etiqueta en cuestión en la traza.

Rollback

Por último, en esta pestaña se encuentran los controles que permiten deshacer eventos. Esta pestaña estará disponible tanto si se ha cargado una traza como si no. Estas acciones se podrían considerar como análogas de las de la pestaña anterior, es por esto que en este caso cada acción consume elementos del histórico (*History*) y añade elementos a la traza (*Log*). Las acciones disponibles son:

- *Roll steps*, realiza lo mismo que la acción *Replay steps*, solo que en este caso de acuerdo a las reglas de la Figura 5.2 y siguiendo el histórico del cuadro *History*.
- *Roll spawn*, deshace la creación del proceso con el *pid* indicado, deshaciendo a su vez todos los pasos que dependan de este, incluyendo todos los del histórico de dicho proceso. Siempre que exista un proceso con dicho *pid* en el histórico.
- *Roll send*, deshace el envío del mensaje con la etiqueta indicada, y a su vez la recepción de dicho mensaje. Siempre que exista un mensaje con dicha etiqueta en el histórico.

- *Roll receive*, deshace la recepción del mensaje con la etiqueta indicada, siempre que exista un mensaje con dicha etiqueta en el histórico.
- *Roll variable*, deshace la creación de la variable con el nombre indicado, siempre que exista en el contexto actual de algún proceso. En caso de que existan varios procesos donde se defina una variable con dicho nombre, tendrán prioridad los procesos cuyo *pid* sea menor.

6.2.4. Información del sistema

La información relativa al sistema está situada en la esquina inferior derecha, bajo el título *System Info*, y consta de dos elementos:

- *Mail* muestra el contenido del buzón global, donde se encuentran los mensajes que han sido enviados pero que todavía no han sido recibidos.
- Un cuadro en la parte inferior con dos pestañas:
  - *Trace* muestra un registro de todas las acciones que se han realizado sobre todos los procesos; se podría decir que es una especie de traza que se genera a medida que se ejecuta el código.
  - *Roll log* muestra un registro de todos los eventos que se realizan cuando se ejecuta una acción de pestaña *Rollback*; esto sirve para ver qué otras acciones se han realizado sobre otros proceso de forma fácil.

6.3. Uso

Para iniciar una sesión de depuración primero hay que cargar el fichero con el código fuente del programa que se desee depurar. Esto se realiza mediante la opción *Open* bajo el menú *File*, o mediante el uso del atajo de teclado `Ctrl+O`. Una vez cargado el código fuente, se puede iniciar la ejecución de programa de dos formas:

- Seleccionando una función como punto de entrada al programa, especificando los argumentos para la función y pulsando el botón *Start*.
- Cargando una traza de una ejecución anterior, mediante la opción *Load Trace* del menú *File*, o mediante el atajo de teclado `Ctrl+T`.

Una vez iniciada la sesión de depuración se habilitarán las acciones correspondientes, bajo el panel *Actions*, dependiendo de si se trata de una sesión normal o de *replay*.

6.3.1. Ejemplo

A continuación, se muestra un ejemplo de una sesión de depuración sobre un programa que no funciona como esperamos, junto con todos los pasos que se siguen hasta encontrar el error. Para este ejemplo se va a usar el programa *sleeping barber*.[5]

[5] El código fuente del programa *sleeping barber* está disponible en: `https://github.com/mistupv/cauder-v2/tree/master/case-studies/barber`

```
$ erl
1> c(barber).
{ok,barber}
2> barber:main().

Customers: John and Joe

'John' finished
'Joe' finished
stop
3>
```

Figura 6.3: Ejecución normal (correcta) del programa *sleeping barber*

En la Figura 6.3 se muestra la salida que se espera del programa *sleeping barber*.

```
$ erl
1> c(barber).
{ok,barber}
2> barber:main().

Customers: John and Joe

'John' finished
```

Figura 6.4: Ejecución real (errónea) del programa *sleeping barber*

No obstante, el comportamiento real del programa es diferente, tal y como se muestra en la Figura 6.4, donde pasados unos instantes podemos concluir que se ha producido un *deadlock*, por lo que pasamos a usar el depurador para buscar qué lo ha causado.

```
$ make load
1> tracer:trace("barber:main()", self(), [{dir,"examples"},{log_dir
    ,"barber_results"}]).
Instrumenting..."examples/barber.erl"

Customers: John and Joe

'John' finished
{[{90,spawn,92},
  {90,spawn,93},
  {90,spawn,...},
  {92,...},
  {...}|...]}
2>
```

Figura 6.5: Generación de la traza del programa *sleeping barber*

Primero tenemos que generar una traza para usar en el depurador. Esta se puede obtener haciendo uso del programa *tracer*[6], tal y como se muestra en la Figura 6.5.

[6]El programa *tracer* está disponible en `https://github.com/mistupv/tracer`

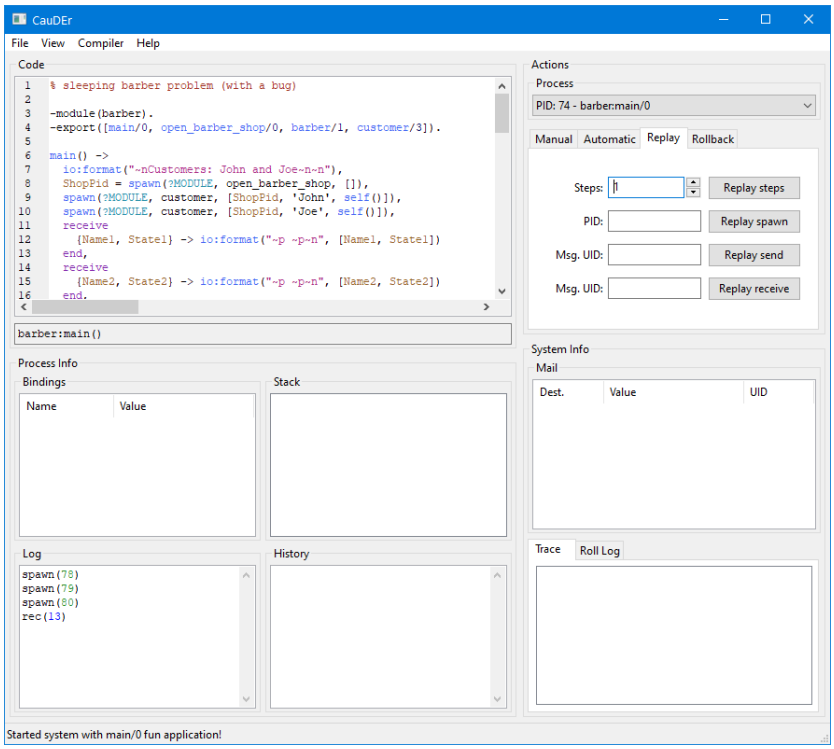

Figura 6.6: Ventana principal de CauDEr al inicio de la sesión de depuración

Ahora ya podemos iniciar CauDEr 2.0. Una vez iniciado el depurador deberemos cargar el código fuente del fichero `examples/barber.erl` y a continuación la traza que hemos generado previamente, la cual estará en la carpeta `barber_log`. Tras validar estos pasos, la ventana del depurador debería tener un aspecto similar al de la Figura 6.6. A continuación, se enumeran los pasos a seguir hasta encontrar el error:

1. Avanzamos 100 pasos en el proceso principal (PID: 74), con esto se habrán creado todos los procesos necesarios (la tienda, el barbero y los dos clientes). Algunos de estos procesos tendrán trazas que todavía no están vacías pero no importa.

2. Deshacemos el envío del mensaje con UID 1, lo cual desencadenará que se deshagan varias acciones debido a la dependencia causal que existe entre ellas. Estas acciones aparecerán en la pestaña de *Roll log*.

3. Repetimos un paso en la tienda (PID: 78), en el cual se debería haber recibido un mensaje con el cliente "Joe".

4. Ahora nos centramos en el proceso de la tienda (PID: 78) y su traza, y comprobamos que todo se desarrolla con normalidad:

    4.1. Debería enviar el mensaje "wakeup" al barbero, es decir, repetir el envío del mensaje con UID 4.

4.2. Debería recibir el mensaje "ready" del barbero, es decir, repetir la recepción del mensaje con UID 2.

4.3. Debería enviar el mensaje con el cliente "John" del barbero, es decir, repetir la recepción del mensaje con UID 6.

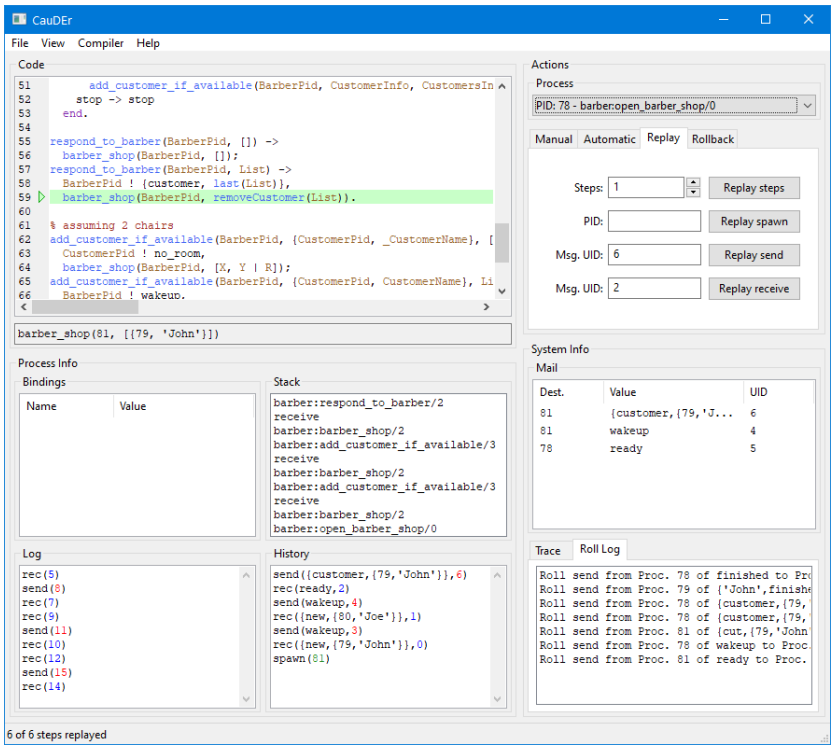

Figura 6.7: Ventana principal de CauDEr al inicio de la sesión de depuración

4.4. Avanzamos ahora paso a paso hasta el siguiente ciclo de `barber_shop`. Tras avanzar tan solo 6 pasos vemos que la siguiente llamada es `barber_shop(81, [{79, 'John'}])`, tal y como se puede observar en la Figura 6.7, lo cual es incorrecto. Hemos enviado a "John" al barbero pero hemos eliminado a "Joe" de la sala de espera.

Para solucionar este error habría que corregir la implementación de la función `removeCustomer` para que en vez de borrar el primer elemento de la lista que se le pasa, borre el último elemento.[7]

[7] La versión corregida del programa *sleeping barber* está disponible en `https://github.com/mistupv/cauder/blob/master/case-studies/barber/barber_fixed.erl`

Capítulo 7

Conclusión y trabajo futuro

En este proyecto hemos diseñado e implementado una nueva versión del depurador CauDEr, un depurador reversible para programas Erlang. La versión original del depurador consideraba programas en formato Core Erlang. Más concretamente, en este trabajo hemos definido la semántica reversible al nivel de Erlang (en vez de Core Erlang) y hemos reimplementado el depurador en base a la nueva semántica. Además, se han integrado las dos aproximaciones anteriores, la que permitía explorar una ejecución de forma libre y la que se basaba en una traza de ejecución. Todo ello ha cambiado significativamente a mejor la usabilidad del depurador.

Con este trabajo se ha completado el primer paso para obtener un depurador (reversible) completamente funcional, sin embargo todavía quedan muchas mejoras y posibles actualizaciones, algunas de la cuales se presentan a continuación, sin ningún orden en particular:

- Permitir iniciar una sesión de depuración en modo manual, es decir, sin indicar una traza, ya que actualmente solo se puede usar en modo *replay*.
- Añadir acciones menos *detalladas*, que permitan avanzar a la siguiente expresión completamente, en lugar de ir aplicando reglas individuales como ocurre actualmente.
- Ampliar la cantidad de construcciones soportadas, añadiendo soporte para, por ejemplo, excepciones o expresiones `try ... of ... catch ... end`.
- Añadir soporte para entrada/salida, es decir, llamadas a funciones del módulo `io`.
- En la implementación, desacoplar completamente la lógica de la interfaz, lo que permitiría un desarrollo más independiente de ambas partes, así como la posibilidad de usar otras bibliotecas para la interfaz, si fuese necesario.
- Añadir soporte para programas distribuidos.

Bibliografía

[1] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[2] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[3] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, 2018.

[4] Ivan Lanese, Adrián Palacios, and Germán Vidal. Causal-consistent replay debugging for message passing programs. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11535 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2019.

[5] Robert H.B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[6] Naoki Nishida, Adrián Palacios, and Germán Vidal. A reversible semantics for erlang. In Manuel Hermenegildo and Pedro López-García, editors, *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, volume 10184 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2017.

[7] Naoki Nishida, Adrián Palacios, and Germán Vidal. Reversible computation in term rewriting. *J. Log. Algebraic Methods Program.*, 94:128–149, 2018.

[8] Hans Svensson, Lars-Ake Fredlund, and Clara Benac Earle. A unified semantics for future erlang. In *9th ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010.