UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

Departamento de Sistemas Informáticos y Programación

SEMÁNTICAS FORMALES PARA UN LENGUAJE FUNCIONAL PARALELO

MEMORIA PRESENTADA PARA OPTAR AL GRADO DE DOCTOR POR

Mercedes Hidalgo Herrero

Bajo la dirección de la Doctora:

Yolanda Ortega Mallén

Madrid, 2004

ISBN: 84-669-2594-5

Semánticas Formales para un Lenguaje Funcional Paralelo

Tesis Doctoral

Mercedes Hidalgo Herrero

Departamento de Sistemas Informáticos y Programación

Facultad de Ciencias Matemáticas

Universidad Complutense de Madrid

Marzo 2004

Semánticas Formales para un Lenguaje Funcional Paralelo

Memoria presentada para obtener el grado de
Doctora en Ciencias Matemáticas

Mercedes Hidalgo Herrero

Dirigida por la profesora

Yolanda Ortega Mallén

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid
Marzo 2004

A los que partieron deseando
ver el fin de este sueño


PERU
BRASIL
OCÉANO
PACÍFICO
Lima
Jauja

"Turing comprendió las dificultades que afrontaba al intentar explicar cómo se transformaba por arte de magia la sintaxis en semántica; durante un momento fijó la mirada en la lluvia, que ahora golpeaba con más fuerza que nunca contra los cristales de las ventanas, y se agitó en su silla un poco desconcertado tanto por la confusión de Snow como por la intensidad de la pregunta de Haldane. ¿Cómo se puede *explicar* científicamente un instinto visceral o una convicción firme? se preguntó a sí mismo. ¿Qué clase de argumentos lógicos puedo dar a un materialista hostil como Haldane, o Schrödinger, para convencerles de que la inteligencia es simplemente cuestión de seguir el tipo adecuado de reglas? Diga lo que diga, seguro que Wittgenstein argumentará en mi contra hasta su último aliento. ¿Por qué acepté venir esta noche a esta reunión? La situación es verdaderamente desesperada. Pero ahora estoy metido hasta el cuello, así que supongo que no hay más remedio que echarse al ruedo y esperar lo mejor."

John L. Casti. *El Quinteto de Cambridge*. 1998.

Abstract

The aim of this work is to define formal semantics for a language with the main characteristics of the functional parallel language Eden, which is an extension of the most well-known lazy functional language, Haskell, since Eden employs generic process definitions that, when instantiated, create processes dynamically. In Eden, the parallelism is explicit and communications are implicit, whereas the non-determinism is explicit via a predefined process abstraction. All these characteristics have been gathered in a kernel language much simpler than Eden, Jauja, and the formal semantics have been defined for it.

Firstly, we define an operational semantics whose level of abstraction is far from the mechanisms of any virtual machine. Our basis is the *natural semantics by Launchbury*, extended by Baker-Finch et al. to define a formal semantics for the parallel functional language GpH (Glasgow Parallel Haskell), also defined on top of Haskell but with semi-explicit parallelism. In the case of Jauja, we extend the latter semantic model in order to represent processes. Using the new operational semantics, we define measures for studying the efficiency of programs and whether the parallelism has been fruitful or not.

Secondly, and on behalf of language users, we present a denotational semantics which defines the languaje formally but with a higher level of abstraction. However, the denotational model is not a direct denotacional semantics, but it is based on continuations; in this way we can express the laziness of Jauja and the side-effects resulting from an expression evaluation: process creation and communications. All these side-effects are gathered in a continuation.

Finally, we use the denotational-continuation model to define the meaning of pH (Parallel Haskell) and GpH. Previously, both of them have been defined via operational semantics, but the latter are so different that a comparison among pH, GpH, and Jauja was unfeasible. On the other hand, these three languages are defined on top of Haskell, but using three different approaches of parallelism: implicit for pH, semi-explicit for GpH, and explicit for Jauja. A common semantic framework (in this case, based on continuations) allows us to compare these three approaches.

Resumen

El objetivo del trabajo es definir semánticas formales para un lenguaje que contiene las principales características del lenguaje funcional paralelo Eden, extensión del lenguaje funcional perezoso por excelencia, Haskell, pues Eden permite definir esquemas genéricos de proceso y crear dinámicamente ejemplares a partir de los mismos. Así como Eden presenta *paralelismo explícito*, la *comunicación* es, en cambio, *implícita*. En cuanto a la introducción del no-determinismo, es explícita y se limita a la invocación de un proceso predefinido. Estas características las hemos incluido en un lenguaje más simple, Jauja, y hemos procedido a definir semánticas formales para este último.

La primera semántica tratada es la operacional, con un nivel de abstracción separado de las particularidades de cualquier máquina virtual. Nos dirigimos hacia enfoques basados en la *semántica natural de Launchbury*, como la semántica operacional que Baker-Finch y otros definieron para el lenguaje funcional paralelo GpH (Glasgow Parallel Haskell), también definido sobre Haskell pero empleando paralelismo semi-explícito. Para Jauja extendemos este modelo operacional para poder representar procesos. A partir de la semántica operacional construida, se definen medidas para estudiar la eficiencia de los programas, sobre si la introducción del paralelismo ha sido fructífera o no.

Pensando en un usuario del lenguaje, hemos definido una semántica denotacional que refleja los resultados de la evaluación sin bajar tanto el nivel de abstracción. No obstante, el modelo denotacional elegido no es una semántica denotacional directa, sino un modelo denotacional de continuaciones para poder expresar la pereza de Jauja y los posibles *efectos laterales* producidos como resultado de la evaluación de una expresión: creación de procesos y comunicaciones subyacentes. La consideración de estos efectos laterales estará implícita en una continuación.

En el trabajo también empleamos el formalismo denotacional de continuaciones para definir el significado de pH (Parallel Haskell) y de GpH. Para ambos lenguajes ya se habían definido semánticas operacionales, pero las diferencias de base de estas semánticas no permitían una comparación fácil entre los algoritmos escritos en pH, los implementados en GpH y los programados en Jauja. Por otra parte, estos tres lenguajes son representantes de los tres enfoques fundamentales de introducción de paralelismo en un lenguaje funcional, en el caso que nos ocupa Haskell: pH de paralelismo implícito, GpH de paralelismo semi-explícito y Jauja de paralelismo explícito. De modo que un mismo marco semántico (de continuaciones) nos permite comparar también estos tres paradigmas.

Agradecimientos

> Ingrato es quien niega el beneficio recibido;
> ingrato, quien no lo restituye; pero de todos,
> el más ingrato es quien lo olvida.
>
> Lucius Annaeus Seneca

No quisiera dejar pasar la oportunidad de mostrar mi agradecimiento a todas aquellas personas que, en algún momento de este largo camino, me han acompañado y servido de apoyo científico y/o moral.

El lento paseo se vio dificultado durante un tiempo por una cuesta que parecía insalvable. Cuando todo indicaba que el cambio de rasante no llegaba, apareció David de Frutos. Él me hizo llegar arriba y divisar un paisaje denotacional esplendoroso en contraposición con el que yo hasta entonces veía. La presencia de David también ha sido constante en los últimos coletazos de este trabajo: gracias por el tiempo que durante las últimas semanas me has dedicado.

La constante disponibilidad que Fernando Rubio y Alberto*e* de la Encina me han brindado fue de gran ayuda en la fase de implementación: el primero colaboró en mi transformación en una persona *funcional* y el segundo introdujo más aleatoriedad en mi mundo. Pero sus ayudas no quedaron ahí, pues Fernando siempre ha cogido el teléfono para responder preguntas y me ha enviado todo lo posible para orientarme; por su parte, Alberto durante las últimas semanas estuvo deseando que llegara el momento de imprimir y encuadernar este trabajo para poder ayudarme con ello.

De mi mente nunca desaparecerán los días de estancia en la Heriot-Watt University. En esos días, Kevin Hammond, Hans-Wolfgang Loidl, Greg Michaelson y Phil Trinder me obsequiaron con muy fructíferos y acertados comentarios.

En la misma línea, no puedo pasar por alto las conversaciones mantenidas con los miembros de Eden: todos ellos han contribuido a que esta tesis tuviera su razón de ser y a que yo viera con otras gafas cada propuesta de semántica. Por ello, ni puedo ni quiero dejar de agradecer su colaboración a Alberto de la Encina, Luis Antonio Galán, Rafael

Martínez, Manuel Núñez, Yolanda Ortega Mallén, Pedro Palao, Cristóbal Pareja, Ricardo Peña, Fernando Rubio y Clara Segura. Asimismo, deseo hacer este agradecimiento extensivo a Rita Loogen, impulsora del grupo Eden de la Philipps-Universität Marburg y emisora de comentarios siempre reconfortantes.

Y cuando para un mismo concepto se utilizan diferentes definiciones dependiendo del ámbito de uso, es imprescindible emplear la adecuada en cada caso. En esta tarea fue fundamental la funcionalidad de Natalia López, que, además, siempre es capaz de cambiarme cara y ánimo con su "hola" telefónico.

No puedo olvidar las vísperas de congresos y la constante presencia de Olga Marroquín para pelearse con las transparencias y la impresora. Además es inestimable su resumen de los trámites y papeleos necesarios para poder presentar este trabajo. Igualmente le agradezco el tiempo de trabajo con su "jefe" que le he robado.

A Martín M. Garbayo le estaré siempre agradecida por enseñarme el concepto de "socialización de libros": yo llevo con su manual de LaTeX socializado tanto tiempo que ya dudo que él recuerde que es su propietario. Asimismo, desde que le conozco ha mostrado su ferviente deseo de que yo terminara la tesis. Lo mismo he de agradecer a Juan Miguel Belmonte, Mª del Carmen Chamorro, Mª del Sagrario Simarro y Francisco Vecino.

Hay dos personas que, cada vez que han hablado conmigo en el último año, me han inyectado energía moral para terminar: Luis González y Juan José Pulido. Y dentro del apartado de impulso anímico, mención especial merecen Manuela Vega e Ignacio Sánchez. A este último, además, deseo agradecerle mi primer *Dra.* y el significado que él incluyó en esa denominación.

Por supuesto, quiero incluir entre los agradecidos a mi familia, por aguantar desesperaciones, agobios y faltas de tiempo para estar con ellos. ¡Ah, Bea!, y por Granada. Y al grupo extendido de licenciatura, por el aguante que han demostrado ante la falta de correos a la que les tenía sometidos: pasaba mi tiempo escribiendo esta tesis.

El final de este trabajo, un tribunal. Quisiera agradecer a cada uno de los miembros del *mío* su complacencia para formar parte de él y el tiempo y el esfuerzo que empleen en la lectura de esta memoria.

Finalmente, pero sólo por el lugar que ocupa en estos agradecimientos, incluir a mi directora: Yolanda Ortega Mallén. En realidad es la persona a la que más tengo que agradecer. Todo comenzó porque ella así lo deseó, cuando allá por el verano de 1996 ella me expresó su deseo de que participara en sus proyectos de investigación. Mi fluir a su lado continuó con su inestimable orientación durante todos mis estudios de doctorado, y desembocó en el comienzo de esta tesis. Durante el desarrollo de la misma creo que puedo asegurar que ha sido una de las mejores directoras que un doctorando puede tener, por su capacidad de orientación, de trabajo, por su constancia, insistencia, perseverancia, ánimos... Pero sobre todo porque, tras todos estos años, para mí es mucho más que mi directora de tesis.

Índice general

Índice de figuras

CAPÍTULO 1

Prólogo

> El que logra empezar un camino lo tiene ya medio hecho.
>
> Lucius Annaeus Seneca

A lo largo de la Historia, la Humanidad ha tenido sueños que durante mucho tiempo fueron inalcanzables. Famosa es la aspiración de imitar a las aves y planear por el cielo como ellas. Durante siglos muchos intentos se hicieron —véanse por ejemplo los diseños de Leonardo da Vinci— pero hasta el siglo pasado no se consiguió realizar el sueño de volar.

Otro de los anhelos perseguidos ha sido la construcción de un "aparato" que se comportara como el ser humano. En este caso como mejor se ha materializado el sueño es en la ficción, mediante películas futuristas con androides pululando por las galaxias. En el mundo real se diseñaron calculadoras, se automatizaron telares y, desde los años 50, se construyen computadoras, algunas de las cuales han sido diseñadas de tal manera que pueden competir con los ajedrecistas de mayor prestigio mundial. Sin embargo, no todos los ordenadores tienen un propósito tan específico, sino que se pretende que puedan satisfacer numerosas necesidades de la sociedad actual. Y si las máquinas tienen fundamentalmente la misma arquitectura subyacente, ¿qué hace que puedan servir para diferentes objetivos? La respuesta está en los algoritmos que implementan las distintas tareas que con ellas se desea realizar.

Según [Rea95], un algoritmo es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. Una receta de cocina sería un ejemplo de algoritmo: es una secuencia ordenada de operaciones que permiten resolver el problema de

comer. Sin embargo, ¿sabría una computadora preparar una ensalada de pasta siguiendo las palabras textuales de Adriá, Arguiñano o Arzak? Hoy por hoy parece imposible. ¿Cómo hacer que estas máquinas obedezcan nuestros deseos? Parece que el primer paso, que ya se dio hace muchos años, fue el de establecer lenguajes comunes entre la máquina y el ser humano: los lenguajes de programación. En un principio estos lenguajes estaban más cerca de la máquina y más lejos de la persona. Pero el devenir de los años ha tornado esta situación, y ahora los lenguajes se acercan más al modo de pensar de los humanos.

Una de las características del pensamiento humano es el razonamiento. Y es el razonamiento la base del pensamiento matemático. ¿Por qué no acercar los lenguajes de programación al pensamiento matemático? Esta pregunta condujo a diseñar lenguajes de programación cercanos a las definiciones matemáticas.

Una definición clave en matemáticas es la de *relación*. Si nos restringimos al caso binario, la podemos definir a partir de dos conjuntos cualesquiera A y B: una *relación binaria* entre A y B es un conjunto R de pares ordenados contenido en el producto cartesiano de A por B, i.e. $R \subseteq A \times B$; si un par se encuentra en esta relación lo denotaremos por $x\,R\,y$. Se define el *dominio* de la relación R como el conjunto

$$\mathsf{dom}(R) = \{x \in A \mid \exists y \in B . x\,R\,y\}$$

y la *imagen* de R como el conjunto

$$\mathsf{img}(R) = \{y \in B \mid \exists x \in A . x\,R\,y\}$$

Relaciones hay de muchas clases, de equivalencia, de orden, correspondencias, etc. Las de equivalencia definen, en parte, las clases de objetos de los lenguajes de programación orientados a objetos. Pero en este trabajo las relaciones que nos van a interesar son las relaciones que denominamos *funciones*: una relación binaria entre A y B es una *función* si cumple que:

$$\forall x \in \mathsf{dom}(R) \text{ existe un único } y \in B \text{ tal que } x\,R\,y.$$

Las funciones se denotan por f, g, etc., y $x\,R\,y$ se denota por $f(x) = y$. Una función f puede ser *total*, si $\mathsf{dom}(f) = A$, o *parcial*, si $\mathsf{dom}(f) \subsetneq A$.

Por último, las funciones no son entes aislados sino que se relacionan entre ellas. La forma de hacerlo es vía la *composición funcional*: dada una función f entre A y B y otra función g entre B y C, se define f *compuesta con* g como el conjunto

$$g \circ f = \{(x, z) \mid x \in \mathsf{dom}(f) \wedge f(x) \in \mathsf{dom}(g) \wedge g(f(x)) = z\}.$$

De la definición de la composición de funciones se deduce que para poder calcular $g \circ f$ se tiene que cumplir que $\mathsf{img}(f) \subseteq \mathsf{dom}(g)$.

Llegados a este punto resumamos el objetivo: entendernos con las computadoras empleando un lenguaje cuya base sea el concepto matemático de función. ¿Es esto posible?

No es que sea posible, es que este objetivo ya se cumplió entre 1959 y 1960 con el desarrollo de LISP (List Processing) [MAE+62], el primer lenguaje funcional. El cálculo computacional base de la programación funcional es el λ-cálculo, en el que subyace la notación funcional que hemos visto. Este cálculo fue desarrollado por Alonzo Church [Chu41] en los años 40. El λ-cálculo ha servido de herramienta fundamental para el estudio de los lenguajes de programación, no solamente funcionales, y de su semántica. Además, es la filosofía con la que fueron diseñados lenguajes como LISP [MAE+62], ML [MTH90] o Haskell [PH99, Pey03]. La notación que emplea, ejemplificada para la función identidad $f(x) = x$, es $\lambda x.x$; y cuando esta función se aplica a un elemento de su dominio, por ejemplo $f(3)$, en el λ-cálculo se escribe $(\lambda x.x)\,3$. En general, su sintaxis es la siguiente:

E	$::=$	x	variable
	$\mid$	$\backslash x.E$	λ-abstracción
	$\mid$	$E_1\,E_2$	aplicación funcional

La idea subyacente en el paradigma funcional es disponer de una colección de funciones básicas a partir de las cuales se pueden definir todas las funciones que se desee. Por ejemplo, se puede disponer de la función mayor o igual, $\geq$, ya definida y emplearla para definir la función mínimo:

$$\mathsf{min}(x, y) \begin{cases} y, & \text{si } x \geq y; \\ x, & \text{e.o.c.} \end{cases}$$

Y a partir de la función mínimo podemos definir una función que devuelva cuál es el menor de tres números:

$$\mathsf{menor}(x, y, z) = \mathsf{min}(\mathsf{min}(x, y), z).$$

Y utilizando menor, calcular el menor de seis números:

$$\mathsf{menorSeis}(a, b, c, d, e, f) = \mathsf{min}(\mathsf{menor}(a, b, c), \mathsf{menor}(d, e, f)).$$

Siguiendo la línea descrita, con una colección suficientemente rica de funciones básicas se pueden definir cómodamente nuevas funciones por medio de la composición funcional. Según [Hen80], entre las funciones básicas se deben incluir las aritméticas básicas: suma, resta, producto y división. En Haskell, por ejemplo, con ayuda de las funciones básicas se definen todas las de su preludio y a partir de ahí el programador define las suyas propias. En el caso del λ-cálculo se va más allá, ya que permite definir todas las funciones básicas a partir de la sintaxis mostrada anteriormente, con la que se pueden definir todas las funciones computables.

Sin embargo, como ya se ha indicado, la idea de función no es novedosa, como tampoco lo es la de la programación funcional. No obstante, este paradigma no se ha estancado, sino que se han incorporado cambios en los lenguajes funcionales, entre otros motivos para que sean más eficientes. Uno de estos cambios es el paralelismo. Por ejemplo, los lenguajes Eden [BLO94, BLOP96b], GpH (Glasgow Parallel Haskell) [THM+96] y pH

(Parallel Haskell) [NA01] constituyen ejemplos de lenguajes funcionales paralelos definidos sobre el lenguaje funcional Haskell; y lenguajes como Facile [GKK+93] o CML [Rep93] son lenguajes concurrentes definidos sobre el lenguaje funcional ML.

A partir de aquí nuestro interés se va a centrar fundamentalmente en Eden [BLOP96b]. Este lenguaje surge por la colaboración entre dos grupos de investigación, con sedes en la Universidad Complutense de Madrid y en la alemana Philipps-Universität Marburg, con el objetivo de introducir en un lenguaje funcional expresiones para ser evaluadas en paralelo; el lenguaje funcional fue Haskell, y el paralelo resultante Eden.

El lenguaje natural y los lenguajes de programación tienen características comunes, como el hecho de disponer de una sintaxis que determina cómo se pueden combinar las expresiones de las distintas categorías sintácticas para dar lugar a expresiones más complejas. Sin embargo, por mucho que un diccionario, por ejemplo, de castellano, determine el significado de cada vocablo, sucede en numerosas ocasiones que una oración tiene más de una interpretación posible. La conjunción de este hecho con la redacción de manuales de lenguajes de programación en lenguaje natural, deriva en la implementación de distintos compiladores para un mismo lenguaje que no producen ejecuciones equivalentes. Esta posibilidad no debe tener cabida en un lenguaje de programación, y el medio para no permitir estas polisemias es la definición de semánticas formales.

Una semántica formal define de manera rigurosa y completa un lenguaje de programación, ofreciendo una definición matemáticamente precisa y sin ambigüedad posible. Pero los usos de una semántica formal van más allá de la propia definición del lenguaje:

- Las semánticas formales son una herramienta para demostrar propiedades sobre el comportamiento de los programas o la equivalencia entre éstos. Asimismo, pueden emplearse para establecer propiedades del lenguaje que definen.

- La verificación de programas, o demostración de su corrección con respecto a la especificación que se quiere que satisfagan, es posible gracias a la definición formal de su semántica.

- Una definición formal constituye una guía para diseñadores de lenguajes, pues su tarea no es sencilla y con la ayuda de una semántica formal se pueden diseñar lenguajes complejos pero que son equivalentes, de alguna manera, a una estructura matemática simple y natural.

- Existe herramientas que toman como entrada una semántica formal definida para un lenguaje y le "dan vida", de manera que el usuario puede ejecutar programas en su lenguaje y observar cómo se comportan. De este modo, en la fase de diseño del lenguaje se pueden experimentar diferentes elecciones semánticas y observar las diferencias que éstas conllevan.

- Además, siguiendo las directrices de la semántica formal se pueden implementar compiladores e intérpretes del lenguaje. Cierto es que éstos no son suficientemente eficientes como para que tengan un uso práctico, pero sí que permiten tanto al implementador del lenguaje como al usuario programador del mismo estudiar de

manera práctica el comportamiento de las expresiones según la semántica que define el lenguaje.

Por un lado hemos presentado brevemente los lenguajes funcionales; a mitad del camino mencionamos que los lenguajes funcionales podían ser ampliados para tratar con el paralelismo; por otro lado hemos descrito lo que es una semántica formal. De esta forma ya tenemos todos los elementos base de este trabajo, ya que nuestro principal objetivo es definir formalmente la semántica de un lenguaje que contiene las principales características del lenguaje funcional paralelo Eden. Teniendo este objetivo general en mente, hemos estructurado esta memoria en las siguientes partes:

Preliminares: en esta parte se presentarán con mayor profundidad los dos conceptos fundamentales de este trabajo, a saber, los lenguajes funcionales paralelos y las semánticas formales. En el Capítulo 2 se estudiará la conjunción del paradigma funcional y del paralelismo: en primer lugar se discutirán brevemente las ventajas e inconvenientes del uso de lenguajes funcionales; seguidamente se expondrán diferentes enfoques de paralelismo; para concluir, se analizarán diferentes opciones de mezclar programación funcional y paralelismo. En el Capítulo 3 se recorrerán las diferentes formas de definir formalmente la semántica de los lenguajes de programación, haciendo una escala en las que se han utilizado para definir lenguajes funcionales concurrentes y paralelos. Finalmente, en el Capítulo 4 se describirá Jauja, un minilenguaje en el que incluiremos las principales características de Eden y del que definiremos su semántica en el resto del trabajo.

Semántica operacional: esta parte del trabajo se concentrará en el desarrollo de una semántica operacional para el lenguaje introducido en el Capítulo 4. En [BLOP96b] ya se definió una semántica operacional para Eden. Esta semántica se basaba en los cambios que la evaluación de una expresión producía en el estado de una máquina abstracta. Nuestro interés reside en definir una semántica operacional con un nivel de abstracción separado de las particularidades de cualquier máquina virtual. Por ello nos dirigiremos hacia enfoques basados en la *semántica natural de Launchbury* [Lau93]. En nuestra investigación previa hemos centrado nuestra atención en la semántica operacional que en [BKT00] se definió para GpH. Ambas semánticas serán detalladas en el Capítulo 5. En el Capítulo 6 extenderemos este modelo operacional que describe GpH con el objeto de poder definir los procesos en Jauja/Eden. A partir de esta semántica operacional definiremos medidas que darán una visión sobre la eficiencia de los programas, y sobre si la introducción del paralelismo es fructífera o, por contra, solamente consume recursos sin contribuir a la obtención del valor final. En definitiva, será una semántica pensada para implementadores del lenguaje, con las indicaciones operacionales de cómo funciona cada expresión característica de Eden.

Semántica denotacional: una semántica con tantos detalles como los que presenta una operacional no es la más orientada hacia los intereses de un usuario del lenguaje. Por ello, entre nuestros objetivos se encuentra la definición de una semántica denotacional que refleje los resultados de la evaluación sin bajar tanto el nivel

de abstracción. No obstante, el modelo denotacional elegido no será una semántica denotacional directa, sino que será un modelo denotacional de continuaciones [Ten76]. En el Capítulo 7 detallaremos cómo surgió y en qué consiste el concepto de continuación. La meta final de utilizar un modelo de continuaciones es poder definir adecuadamente las características principales de Eden, a saber, pereza (consultar el glosario en el Apéndice B) y paralelismo. Además, pretendemos comparar Eden con GpH y pH por dos motivos: los tres son lenguajes basados en Haskell y cada uno de ellos introduce el paralelismo de un modo distinto. Cierto es que para cada uno de los lenguajes se dispone de semánticas formales operacionales, para Eden la incluida en el Capítulo 6 y la definida en [BLOP96b], para GpH la descrita en [BKT00] y para pH las definidas en [AAA+95, NA01]. Sin embargo, todas estas semánticas difieren ampliamente ya en su base, por lo que para poder realizar comparaciones es conveniente disponer de un mismo marco semántico para los tres lenguajes. Estas razones nos impulsan a definir una semántica formal basada en continuaciones para cada uno de ellos. La del minilenguaje representante de Eden, Jauja, se detalla en el Capítulo 8 y las de los otros dos lenguajes en el Capítulo 9.

Conclusiones y trabajo futuro: terminaremos el trabajo presentando las conclusiones a las que hemos llegado tras la definición de las distintas semánticas formales y perfilando las líneas futuras de investigación.

Además, la memoria se completa con dos apéndices: uno con demostraciones que si se incluyeran en el texto harían más engorrosa la lectura de éste, y un glosario de términos propios del paradigma funcional.

PARTE I

PRELIMINARES

CAPÍTULO 2

Paralelismo en lenguajes funcionales

> Una de las grandes tragedias del ser humano consiste precisamente en que se le mutile ese afán de belleza, en que se impida su desarrollo.
>
> Rosa Montero

Varios son los objetivos que perseguimos con este capítulo. El primero de ellos es analizar el paradigma de programación funcional tanto desde el punto de vista de sus partidarios como desde el de sus detractores, presentando los argumentos con que los primeros defienden dicho paradigma. En [Mor82] se alaban las buenas propiedades de los lenguajes funcionales, pero también se detallan los argumentos con los que sus detractores apoyan su rechazo. Asimismo, se analizará una de las principales mejoras que se han incorporado a estos lenguajes desde el punto de vista de la eficiencia en su ejecución: el paralelismo.

Para ello comenzaremos exponiendo las distintas formas de incluir paralelismo en un lenguaje secuencial y estudiaremos clasificaciones del paralelismo atendiendo a diversos criterios. Para cada forma de paralelismo nos detendremos en ver su introducción en los lenguajes funcionales.

2.1. El paradigma funcional, ¿por qué sí y por qué no?

La mayor parte de los programadores que reniegan del paradigma funcional arguyen que los programas desarrollados con lenguajes funcionales son menos eficientes, hecho

que viene favorecido por la arquitectura imperante de las computadoras existentes: la arquitectura Von-Neumann.[1] Refrenda también esta opinión el alto nivel de abstracción de estos lenguajes, pues éste provoca la pérdida por parte del programador del control directo sobre los recursos de la máquina; digamos que el programador no puede "apañar" una mejor ejecución, forzando el acceso (casi)directo a dichos recursos. Pensemos en lenguajes imperativos y recordemos prácticas como la programación con memoria dinámica, en la que el programador se encarga de reservar zonas de memoria y luego de recoger la basura resultante. Sin embargo, en [Sto82] se alaba este alto nivel de abstracción precisamente porque permite al programador olvidarse de los detalles de lo que la máquina hace en concreto, siendo esta abstracción la que dota a los lenguajes funcionales de sus agradables propiedades, como su cercanía al razonamiento matemático.

En cuanto a su uso, sus detractores aseguran que los lenguajes funcionales no son naturales, a lo que Morris contesta en [Mor82] que tampoco es natural usar tenedor y cuchillo en la mesa o respetar los protocolos diplomáticos y, sin embargo, todos ellos son de uso común en la civilización moderna. Y continúa Morris luchando en contra de la esgrimida falta de naturalidad de los lenguajes funcionales, comparando los programas funcionales con el Haiku,[2] emparejando el sentimiento de gran placer estético que produce la escritura de un poema con el que supone la composición de un programa en un lenguaje funcional. Se puede entonces considerar esta belleza como un argumento a favor. Sin embargo, los programadores "reales" tienen por objetivo la búsqueda de resultados eficientes y rápidos y no se plantean reelaborar un programa que cumple ya su misión, a fin de hacer más placentera su lectura, y, de paso, de generar un programa más fácil de comprender, modificar, mantener, etc. Las prisas de la civilización moderna y las cuotas de productividad a toda costa conducen a que los programadores prefieran lenguajes que les permitan resolver problemas "a la primera", aunque sea de una forma chapucera y difícil de verificar su corrección o modificar los objetivos para cubrir otros nuevos.

Así, el reto más importante de los informáticos que trabajan con el paradigma funcional consiste en lograr que los lenguajes funcionales sean más prácticos y fáciles de usar. Desde el punto de vista teórico, (casi)todos los lenguajes tienen la misma capacidad expresiva,[3] pero también es cierto que se construyen frases con el mismo significado en castellano, chino, francés, inglés, etc. Llegados a este punto, ¿qué es lo que realmente importa? Pues si el significado va a ser el mismo y lo que "molesta" es que las sintaxis sean distintas, unifíquense las sintaxis de los lenguajes funcionales entre sí en la medida de lo posible: empleemos las mismas palabras clave para construir expresiones semánticamente equivalentes. Este es el reto que planteó Morris en [Mor82], pero parece que esta propuesta no ha sido demasiado seguida por los diseñadores de lenguajes funcionales. Sin embargo, sí que esta meta fue alcanzada con Prolog en el caso de los lenguajes lógicos.

Los augurios que en la contraportada de [DGH+82] se hacían acerca de la progra-

[1] Los lenguajes imperativos están diseñados de modo que se ajustan a la arquitectura Von-Neumann.
[2] Breve poema japonés de diecisiete sílabas, dispuestas en tres versos de cinco, siete y cinco sílabas.
[3] En concreto, son Turing-completos.

mación funcional no eran desalentadores: "los lenguajes de programación funcionales están más estrechamente relacionados con las notaciones matemáticas tradicionales que los lenguajes que actualmente se usan en la mayoría de las aplicaciones de computación. Tienen muchas ventajas, que hacen de la escritura de programas una actividad más simple y menos propensa a errores. En el pasado, estos lenguajes eran demasiado ineficientes para ganarse la aceptación generalizada, pero desarrollos recientes del *hardware* han venido a cambiar drásticamente la situación. Los lenguajes funcionales prometen ser *los* lenguajes de programación para el futuro". Sin embargo, el paradigma imperativo prevaleció durante muchos años, para luego venir a predominar los lenguajes orientados a objetos, que en el interior de las clases también son imperativos. Pero seamos conscientes de las contradicciones a las que se ha llegado: los programadores repudian los lenguajes funcionales arguyendo que son de muy alto nivel, y, sin embargo, aceptan los orientados a objetos, cuando su nivel de abstracción también es quizás más antropomorfo, ya que los objetos se aproximan al modo de razonar de los humanos. Y, por otra parte, la eficiencia de algunas implementaciones de Java es incluso peor que la de muchos lenguajes funcionales [NA01].

2.2. Paralelismo en los lenguajes de programación

Un objetivo que se ha mantenido constante a lo largo de la historia de las ciencias de la computación ha sido el interés por incrementar la velocidad de cómputo de los programas. Para ello se han desarrollado nuevas arquitecturas hardware. Sin embargo, ésta no ha sido la única vía para conseguir el objetivo pretendido; también se ha investigado en la orientación que podían tomar los lenguajes de programación. Surgieron entonces los lenguajes concurrentes y paralelos.

Independientemente del paradigma bajo consideración, un lenguaje de programación que explote el paralelismo debe contener mecanismos que permitan llevar a cabo las siguientes tareas:

Identificar el paralelismo: es decir, permitir reconocer en un programa los módulos a paralelizar.

Comienzo y fin: el lenguaje tiene que suministrar las herramientas para poder empezar y finalizar las ejecuciones paralelas.

Coordinación: las componentes en que se divide el programa para su ejecución paralela no son entes aislados, sino que tiene que existir la posibilidad de que interaccionen entre ellas, para todas juntas realizar la misión del programa.

En la presente sección vamos a introducir dos clasificaciones diferentes del paralelismo atendiendo a sendos criterios.[4] En primer lugar se va a tratar el paralelismo desde el punto de vista de la granularidad, o de la cantidad y tamaño de los módulos en que un

[4] Existen otras, pero las elegidas son las más interesantes de cara a la presentación de nuestro lenguaje.

programa es dividido para su procesamiento paralelo. En segundo lugar se tratarán los tipos de paralelismo cuya diferencia radica en la medida en la que el programador es responsable de coordinar dicho paralelismo.

2.2.1. Paralelismo y granularidad

La programación paralela conlleva la obtención de mayor eficiencia y la posibilidad de estructurar los programas en módulos que se ejecutan paralelamente interaccionando entre sí. Sin embargo, no es sencillo determinar la cantidad de módulos o procesos, disponiéndose de dos posibles estrategias al respecto:

Paralelismo de grano fino: las tareas en las que se divide el programa conllevan poco trabajo y, en ocasiones —como tras el análisis de estrictez en lenguajes funcionales [BHA85]— la división puede realizarse de manera automática. De esta forma se divide el programa en las instrucciones atómicas del lenguaje, es decir, esta opción puede implicar una división en módulos demasiado pequeños que tendrán que sincronizar entre ellos, con lo que el coste de planificar su ejecución conjunta puede conducir a una pérdida de eficiencia en lugar de a una mejora.

Paralelismo de grano grueso: la unidad de división es el procedimiento, con lo que la necesidad de sincronización es menor. Como ahora se realiza una división en procesos con mayor cantidad de trabajo asignado, se pierden menos recursos y tiempo en la planificación; sin embargo, la interacción entre los módulos y la necesidad por parte de un módulo de esperar cuando otro tiene que producir datos para él, provocan que la ejecución se ralentice. Además, es más complicado obtener este grado de paralelismo de modo automático.

Esta clasificación quedará inmersa en la siguiente, que, de hecho, es mucho más compleja y extensa.

2.2.2. Paralelismo implícito, explícito y la línea continua que los une

Cuando el criterio de clasificación de los lenguajes paralelos es el grado de libertad que tiene el programador para establecer los puntos del programa en los que desea que se paralelice la ejecución del mismo, es habitual distinguir dos enfoques del paralelismo, a saber, implícito y explícito. Pero si somos más precisos nos encontramos con que éstos son la cota inferior y superior, respectivamente, de todos los enfoques existentes en este marco, y que entre ellos existe una línea casi continua de tipos de paralelismo. Nosotros vamos a centrarnos, aparte de en las cotas mencionadas, en el paralelismo semi-explícito, que situaremos entre ambas.

2.2.2.1. Paralelismo implícito

Este tipo de paralelismo se caracteriza por permitir al programador escribir sus programas sin tener necesidad de preocuparse de la explotación del paralelismo, pues dicha explotación es realizada de manera automática, ya sea por el compilador o en tiempo de ejecución. El paralelismo que se intenta desarrollar es el que proviene de la semántica de reducción o, en el caso de paralelismo de datos, de la semántica de las operaciones especiales para tratar con dichos datos. De este modo, el paralelismo es transparente al programador, como tantos otros elementos de la ejecución de los programas, como la gestión de memoria o la de la unidad central de proceso.

La introducción de este tipo de paralelismo cumple con el objetivo de mejorar la eficiencia de la ejecución, al tiempo que no incrementa la complejidad del desarrollo de *software*, pues el diseño del programa es el mismo que en la programación secuencial.

Sin embargo, no todo puede ser tan perfecto: la extracción del paralelismo inherente a un programa no es tarea sencilla. No obstante, dependiendo del paradigma de programación que consideremos esta tarea es más o menos compleja:

Lenguajes imperativos: su característica fundamental es marcar la secuencia de ejecución de los programas, pero la extracción de paralelismo implícito conlleva la ruptura de dicha secuencia. En consecuencia, solamente se obtienen resultados positivos para un conjunto muy restringido de aplicaciones, como pueden ser la operaciones intensivas sobre algunas estructuras de datos como los *arrays*.

Lenguajes declarativos: su situación de cara a la explotación del paralelismo implícito es más prometedora, primero porque su nivel de abstracción es más elevado, segundo porque se caracterizan por su transparencia referencial (ver el glosario del Apéndice B), tercero porque su semántica operacional se basa en alguna forma de no-determinismo (selección de cláusulas en la programación lógica y la aplicación funcional en los lenguajes funcionales), y, finalmente, porque disponen de esquemas de evaluación impaciente (ver glosario del Apéndice B) que permiten obtener cómputos como flujos de datos, que son muy adecuados para la ejecución paralela. Gracias a todas estas ventajas, la paralelización de lenguajes declarativos ha sido ampliamente tratada, como se describe para el caso de los lenguajes lógicos en [CC94], o como veremos a continuación en el caso de los lenguajes funcionales.[5]

Lenguajes funcionales con paralelismo implícito

La propiedad de Church-Rosser (ver Apéndice B) es la principal fuente de explotación de paralelismo implícito en los lenguajes funcionales. Esto resulta evidente, pues si la forma normal es única independientemente del orden de evaluación de las subexpresiones, éstas pueden ser evaluadas en paralelo.

[5] Una clasificación exhaustiva de los lenguajes funcionales paralelos se realizó en [Loo99]. Nosotros, aún basándonos en ella, restringiremos nuestra clasificación a los tres grupos descritos en la Sección 2.2.2 y, fundamentalmente, a los lenguajes que serán tratados en la parte dedicada a las semánticas denotacionales.

Por otra parte, si consideramos los lenguajes funcionales *perezosos* (ver el glosario del Apéndice B) nos encontramos con que dicha pereza es un handicap de cara a la extracción de paralelismo implícito. La pereza supone que la evaluación de una subexpresión solamente se realiza si su valor es necesario: se trata pues de una evaluación dirigida por la demanda. Esto conduce a una evaluación de naturaleza secuencial. La filosofía de la pereza en la que no se evalúa nada que resulte ser innecesario choca con el hecho de paralelizar todas las subexpresiones posibles. Con el objeto de no desarrollar paralelismo especulativo se realizan diversos análisis de estrictez [BHA85], que detectan qué subexpresiones serán demandadas, con lo que su paralelización no será fuente de paralelismo especulativo.

Dentro de este marco de lenguajes funcionales con paralelismo implícito se encuentra pH [NA01]. En la semántica presentada en [AAA+95] se puede observar que se explota todo el paralelismo derivado de las declaraciones de variables locales, es decir, se evalúan todas ellas aunque no sean necesarias. En los siguientes ejemplos vamos a ver cómo esta evaluación puede ser luego necesaria o simplemente especulativa.

Ejemplo 2.1 *Paralelismo implícito en pH.*

Consideremos la siguiente expresión, por medio de la cual se pretende calcular el producto de dos enteros, siendo cada uno de ellos el resultado de sumar los elementos de sendas listas dadas.

```
let s1 = sum l1,
    s2 = sum l2
in (s1 * s2)
```

La evaluación del producto esperará a que ambas variables locales estén ligadas cada una a un entero. Pero esta evaluación no se realiza de manera secuencial, primero el cálculo del valor de `s1` y luego el de `s2`, sino que la ejecución contiene dos hebras paralelas, una la necesaria para evaluar `s1` y la otra para `s2`. En total, la evaluación de la expresión la llevan a cabo tres hebras en paralelo:

$$hebra_1 \mapsto \texttt{sum l1}$$
$$hebra_2 \mapsto \texttt{sum l2}$$
$$hebra_3 \mapsto \texttt{s1 * s2}\ (espera)$$

En este caso todas las subexpresiones evaluadas en paralelo serían demandadas en una evaluación secuencial.

□

Sin embargo, no siempre sucede como en el ejemplo anterior, siendo pH un lenguaje en el que no se hace nada por evitar el paralelismo especulativo. Este hecho queda reflejado en el siguiente ejemplo.

Ejemplo 2.2 *Paralelismo implícito especulativo en pH.*

Consideremos la siguiente expresión con la que se pretende calcular el cuadrado de un entero que es el resultado de sumar todos los elementos de una lista. En la declaración local se ha incluido una segunda variable `s2` que no es necesaria en absoluto para la evaluación del cuerpo.

```
let s1 = sum l1,
    s2 = sum l2
in (s1 * s1)
```

La evaluación del producto esperará a que la variable local `s1` esté ligada a un entero. La ejecución contiene dos hebras paralelas, una para evaluar `s1` y otra para `s2`. En total, tres hebras en paralelo llevan a cabo la evaluación de esta expresión:

$$hebra_1 \mapsto \texttt{sum l1}$$
$$hebra_2 \mapsto \texttt{sum l2}$$
$$hebra_3 \mapsto \texttt{s1 * s1}\ (espera)$$

Observemos que, a diferencia del caso anterior, la evaluación en paralelo de la segunda hebra es un caso de paralelismo especulativo, pues la suma de los elementos de `l2` no se emplea en la evaluación del cuerpo de la declaración.

□

En ambos ejemplos se evalúan en paralelo expresiones que bajo un supuesto de estricta pereza sólo se evaluarían si fueran demandadas.

Podemos pensar que el paralelismo implícito es la mejor forma de introducir paralelismo en los lenguajes funcionales, pues el programador puede seguir moviéndose en el mismo nivel de abstracción. Sin embargo, a pesar de las facilidades que ofrece la programación declarativa, no todo son ventajas en la explotación del paralelismo implícito, no siempre resulta interesante explotar todo el paralelismo implícito. Por un lado se desconoce el tamaño de los elementos o módulos de cómputo, lo que puede conducir a explotar paralelismo de grano excesivamente fino, introduciendo consecuentemente más lentitud en la ejecución en lugar de incrementar la velocidad. Esta razón lleva a incluir en algunos compiladores un análisis de granularidad y de costes para decidir qué expresiones merece la pena que sean palalelizadas. Por otra parte, pueden darse casos en los que los módulos que se ha decidido paralelizar sólo sean aparentemente paralelizables, siendo de hecho inherentemente secuenciales, lo que conduce a la existencia de muchos puntos de sincronización y a ejecuciones también ineficientes. Además, cuando la cantidad de paralelismo implícito es ingente, como puede ser el caso de los lenguajes funcionales, el proceso de extracción de dicho paralelismo obliga a crear sistemas de ejecución potentes y complejos, lo que puede llegar a ser demasiado lento. Esta razón ha conducido a los investigadores en lenguajes de programación a explotar el paralelismo siguiendo otros enfoques.

2.2.2.2. Paralelismo explícito

Puede decirse que esta manera de explotación del paralelismo es radicalmente opuesta a la anterior: si aquélla era transparente al programador, en ésta es el propio programador quien emplea las construcciones *ad hoc* que le ofrece el lenguaje para describir el comportamiento paralelo deseado para el programa, es decir, se detalla el modo en que el cómputo paralelo se desarrollará.

Con este enfoque se consigue la creación de sistemas reactivos (como pueden ser los sistemas operativos) que, a diferencia de los transformacionales, no tienen porqué devolver resultados, y que consisten en un conjunto de procesos que interaccionan de forma continua con el entorno y, normalmente, exhiben un comportamiento no-determinista.

Las construcciones de paralelismo se han introducido en los lenguajes de programación a diversos niveles. En los primeros lenguajes paralelos los mecanismos eran básicos y de bajo nivel, tratándose de primitivas del tipo `join`, `fork`, semáforos, etc. Conformaban una capa adicional de complejidad que recaía sobre el programador, pues la tarea de programar con estas primitivas puede llegar a ser extremadamente ardua. Con el tiempo se han desarrollado lenguajes en los que los mecanismos o herramientas introducidas para tratar con el paralelismo pasan a ser de mayor nivel de abstracción, incluyendo librerías como PVM [Oak93], con un conjunto de primitivas de comunicación, o lenguajes más sofisticados en los que las expresiones a paralelizar están incluidas en la sintaxis, como Java o como los que se detallarán al tratar de los lenguajes funcionales paralelos.

Pero todo enfoque también presenta ventajas e inconvenientes. Entre las primeras se encuentra la flexibilidad que ofrece al permitir codificar una amplia gama de patrones de ejecución, proporcionando una libertad considerable a la hora de elegir lo que debe ser ejecutado en paralelo y cómo. Sin embargo, presenta el inconveniente de dejar en manos del programador la nada sencilla tarea de gestionar el paralelismo. Inconveniente importante es también el hecho de que deje de ser válida la semántica que se tenía para el lenguaje en su versión secuencial, siendo necesario el desarrollo de una nueva para la versión paralela.

Lenguajes funcionales con paralelismo explícito

El motivo de introducir explícitamente el paralelismo en los lenguajes funcionales ya no es solamente el intento de mejorar la velocidad de ejecución de los programas. Aparece como nuevo objetivo el de aumentar la potencia expresiva del núcleo funcional de estos lenguajes de programación. Ejemplos en los que se ha procedido de este modo son Concurrent Haskell [PGF96], Concurrent ML (CML) [Rep93], Eden [BLOP96b], Erlang [Wik94] o Facile [GKK+93]. Todos estos lenguajes añaden construcciones con las que el programador puede incorporar elementos propios de la programación concurrente. Por ejemplo, en CML se incluyen primitivas que permiten generar hebras paralelas (`spawn`) y declarar canales locales (`chan`). Estas mismas se tienen en Facile, que además incluye no-determinismo explícito. En el caso de Eden, es posible definir abstracciones de procesos y ejemplarizar éstas para crear nuevos procesos; además incluye un proceso especial,

`merge`, que introduce también no-determinismo explícito.

El fin de las construcciones anteriormente mencionadas es poder definir y activar procesos, enviar y recibir mensajes, y sincronizar procesos. Si se introduce no-determinismo, se pierde la transparencia referencial característica del paradigma declarativo. Podría pensarse que entonces estamos dejando de lado dicho paradigma, pero ello no es del todo cierto, pues se siguen manteniendo otras cualidades propias del mismo, como el orden superior o el polimorfismo.

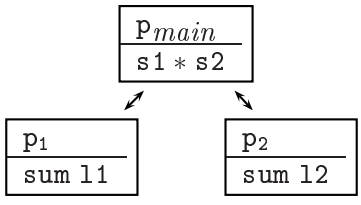
De Eden daremos sobrada cuenta en el Capítulo 4, cuando estudiemos nuestra versión restringida denominada Jauja, pero ilustremos por medio de ejemplos el modo de introducir paralelismo en un programa escrito en Eden.

Ejemplo 2.3 *Paralelismo explícito en Eden.*

Revisamos el Ejemplo 2.1 utilizando ahora la notación de Eden.

```
let p = process list -> sum list,
    s1 = p # l1,
    s2 = p # l2
in (s1 * s2)
```

En este ejemplo la variable `p` se encuentra ligada a una definición o abstracción de proceso. Cuando esta definición sea aplicada a un canal (de entrada) que sea una lista, devolverá como resultado la suma de todos los elementos de la misma. Las variables `s1` y `s2` serán ligadas a sendas salidas de procesos, cada uno de los cuales es un ejemplar creado a partir de la definición `p`. El cuerpo de la declaración realiza la multiplicación de ambos enteros. Observamos que los procesos se crean de manera explícita, lo que no sucedería si no apareciera el operador `#`. La topología de la red de procesos creados, suponiendo que las evaluaciones para `s1` y `s2` no generan nuevos procesos, sería la siguiente:



El proceso principal $\mathtt{p}_{main}$ ha dado lugar a dos nuevos procesos hijos, $\mathtt{p}_1$ y $\mathtt{p}_2$, con los que se relaciona a través de canales de comunicación: el principal envía las respectivas listas a los hijos, y cada uno de ellos devuelve al padre su correspondiente suma.

□

También en Eden con paralelismo explícito es posible la generación de paralelismo especulativo:

Ejemplo 2.4 *Paralelismo explícito especulativo en Eden.*

Veamos un ejemplo con la misma funcionalidad que la dispuesta para pH en el Ejemplo 2.2.

```
let p = process list -> sum list,
    s1 = p # l1,
    s2 = p # l2
in (s1 * s1)
```

En este ejemplo la variable `s2` se encuentra ligada a la creación de un proceso cuyo valor producido no es necesario para la evaluación del valor del proceso principal. Aún así, el proceso se crea, con lo que se trata de un proceso especulativo.

□

En resumen, en los lenguajes funcionales con paralelismo explícito subyace la idea de los procesos comunicantes. El programador tiene el control absoluto sobre estos procesos, pero tiene que definir de manera explícita tanto a ellos como a sus interacciones. A diferencia del enfoque implícito, en este caso la noción de proceso paralelo es explícita, mientras que desde la óptica implícita este concepto no se tenía, como se vio en el caso de pH en los Ejemplos 2.1 y 2.2. Evidentemente se ha perdido el nivel de abstracción altamente confortable de los lenguajes funcionales, pero ello es así porque ahora primamos el deseo de disponer de un tratamiento eficiente de los sistemas reactivos.

2.2.2.3. Paralelismo semi-explícito

Incluimos por último, una tercera clase de paralelismo que denominamos semi-explícito y que sería uno de los enfoques posibles en la línea continua entre el implícito y el explícito. En este caso el programador introduce el paralelismo por medio de anotaciones que indican las partes del programa que se desea que sean evaluadas en paralelo. Estas anotaciones son ignoradas por los compiladores secuenciales y la semántica que se da al lenguaje es la misma que se tenía cuando no se disponía de ellas, es decir, las anotaciones son semánticamente transparentes. Y aunque se tenga un compilador paralelo también es posible que estas anotaciones sean o no obedecidas.

Apoyadas en estas anotaciones se desarrollan estrategias de evaluación —implementadas en procedimientos y funciones concretos— que el programador emplea para influir sobre el comportamiento dinámico de su programa, es decir, para obviar la forma de evaluación por defecto en favor de comportamientos derivados del paralelismo. Asimismo, estas estrategias permiten especificar los puntos deseados de paralelismo.

Una segunda propuesta que enmarcamos en este enfoque es la creación de topologías genéricas de procesos, denominadas *esqueletos* [Col89], que son invocadas desde el programa como si fueran funciones de orden superior y expresan un proceso que el compilador reconoce como tal, de modo que se pueda explotar el paralelismo que conllevan.

Lenguajes funcionales con paralelismo semi-explícito

Finalmente estudiemos las características de los lenguajes funcionales paralelos a caballo entre los implícitos y los explícitos.

Ya dijimos que la cantidad de paralelismo implícito explotable en los lenguajes funcionales podía a ser tan grande que el proceso de extracción podría convertirse en una tarea demasiado costosa. Para solucionar este problema, surgen los lenguajes funcionales en los que el paralelismo es introducido por el programador mediante anotaciones que se tendrán en cuenta en la medida en que los recursos del sistema lo permitan.

Lenguajes funcionales que se han basado en este enfoque son Caliban [Kel89], Concurrent Clean [NSEP91] o Glasgow Parallel Haskell (GPH) [THM+96].

En el marco de la segunda solución propuesta, es decir, en la que considera el uso de topologías genéricas de procesos denominadas *esqueletos*, se incluyen [Col89, DFH+93, HR00, LOP+02].

Detengámonos ahora en un ejemplo de GPH:

Ejemplo 2.5 *Paralelismo semi-explícito en* GPH.

Implementemos con GPH una expresión con la misma intención que la de los Ejemplos 2.1 y 2.3.

```
let s1 = sum l1,
    s2 = sum l2
in s2 'par' (s1 * s2)
```

En este caso se podrán generar dos hebras, una para evaluar `s2` y la otra para evaluar el producto de ambas variables. No obstante, como el orden secuencial de evaluación del producto es primero `s1`, luego `s2` y en tercer lugar el producto, la ejecución real hace que se evalúen en paralelo `s2` y `s1`, para posteriormente calcular el producto. En definitiva, las hebras de ejecución que se crean son dos:

$$hebra_1 \mapsto \texttt{sum l2}$$
$$hebra_2 \mapsto \texttt{s1} * \texttt{s2}$$

la primera evalúa la suma de `l2` y la otra el producto requerido. Nótese que esta generación de hebras solamente tiene lugar si los recursos del sistema lo permiten, pues `'par'` es una mera indicación de paralelismo potencial.

□

De nuevo, el enfoque de paralelismo semi-explícito también puede llegar a generar paralelismo especulativo:

Ejemplo 2.6 *Paralelismo semi-explícito especulativo en* GPH.

```
let s1 = sum l1,
    s2 = sum l2
in s2 'par' (s1 * s1)
```

En este ejemplo de nuevo la suma que produce `s2` no es necesaria para evaluar el valor final del cuerpo, que es el cuadrado de `s1`, por lo que el paralelismo introducido para la evaluación de `s2` es especulativo.

□

Una mejora sobre el paralelismo semi-explícito se describe en [THLP98], donde se conjugan las anotaciones con las estrategias de evaluación, para conseguir así una elevación del nivel de abstracción desde el punto de vista del programador usuario de GPH. Las estrategias se definen sobre las anotaciones de GPH, `seq` y `par`, para especificar el paralelismo, la secuencialidad o el grado de evaluación deseado.

El principal inconveniente de este enfoque es que solamente se ocupa de sistemas transformacionales, no siendo posible definir sistemas reactivos. La solución la vimos al estudiar el enfoque de paralelismo explícito.

CAPÍTULO 3

Estado del arte

> La totalidad de la vida es simbólica porque todo en ella tiene significado.
>
> Boris Pasternak

La definición de semánticas formales para los lenguajes de programación se hace necesaria cuando se pretende establecer, sin ambigüedad posible, el comportamiento de los programas. Los motivos para intentar alcanzar este objetivo son varios, según se indica en [Sto77], y se centran en quién será el destinatario de la semántica:

Implementadores del lenguaje: para que todos los compiladores de un mismo lenguaje sean equivalentes es necesario que todos los implementadores entiendan del mismo modo cada construcción del lenguaje. Con una semántica formal del lenguaje no tienen cabida los equívocos, con lo que todos entenderán exactamente lo mismo para cada expresión sintáctica.

Programadores: éstos emplean las construcciones sintácticas con el fin de escribir programas correctos con respecto a sus especificaciones. Su tarea sería imposible sin una definición exacta del funcionamiento de cada expresión del lenguaje. Esta definición precisa se la proporciona la semántica formal, que dota de significado al lenguaje.

Diseñadores de lenguajes: según Stoy [Sto77], éstos serían los más beneficiados por la utilización de semánticas formales para definir el comportamiento de cada construcción de un lenguaje: "los diseñadores serán guiados hacia el diseño de mejores

(más limpios) lenguajes de programación, con descripciones formales más simples. Y la ventaja de ello será que los programas que se confeccionen, aunque se sigan aplicando los métodos informales habituales, serán probablemente más correctos, porque los programadores serán menos proclives a olvidar la pequeña y crucial excepción de alguna regla general que se aplica en cada caso particular".

3.1. Tipos de semánticas formales

Los modelos básicos tradicionalmente utilizados para dotar de semántica a un lenguaje son: la semántica operacional, la semántica denotacional y la semántica axiomática. Sus diferencias radican en el nivel de abstracción y en el objetivo con el que se define la semántica del lenguaje.

3.1.1. Semánticas operacionales

Este tipo de semánticas concretan un modelo abstracto de ejecución de los programas. En ellas se engloban dos intereses: conocer el resultado que el programa genera y establecer el modo en que lo hace. Tenemos tanto semánticas basadas en máquinas abstractas como otras que, si bien ofrecen un modelo de ejecución, no consideran (explícitamente) ninguna máquina abstracta subyacente.

Un primer enfoque corresponde a la idea de construir una máquina o intérprete formal del lenguaje, con un estado —posiblemente formado por varias componentes— y un conjunto de instrucciones primitivas; definiéndose la semántica por los cambios que esta máquina experimenta durante la ejecución de un programa. Este tipo de semánticas es el ideal cuando se dirigen a los *diseñadores del compilador* del lenguaje. Un compilador traduce un programa escrito en el lenguaje de programación —normalmente de alto nivel— a otro equivalente escrito en un lenguaje de programación más próximo a la máquina. En ocasiones, la traducción produce un programa que será ejecutado por una máquina virtual implementada sobre la máquina real. Esta máquina virtual será muy probablemente la base de la semántica operacional definida. Este es el caso de la máquina DREAM definida para Eden en [BKL+97].

Sin embargo, si se pretende definir una semántica operacional para el uso del *programador*, una opción más adecuada es eliminar la máquina abstracta y establecer la semántica en términos, o bien de pasos de reducción, o bien de reglas, teniéndose en el primer caso una semántica de evaluación y en el segundo una de cómputo, según la clasificación realizada en [Hen90]:

Semántica de evaluación: consiste en una serie de axiomas y reglas para la reducción o ejecución de programas, que dejan de lado la forma concreta en que se realizan los cómputos. Dentro de este grupo se incluyen las que para Concurrent ML se han desarrollado en [PR97, Rep93].

Semánticas de cómputo: evitan los detalles de las máquinas abstractas, pero no se alejan tanto de ellas como las de evaluación, ya que describen los pasos de cómputo necesarios para evaluar una expresión y el orden temporal entre dichos pasos. En este marco se engloban las semánticas descritas en [Fer96, FH99, GMP90], realizadas para lenguajes funcionales concurrentes y paralelos.

Se emplean también las llamadas *semánticas naturales*, que consideran axiomas y reglas de inferencia para caracterizar a las expresiones del lenguaje; a diferencia de las semánticas de evaluación, en este caso la inferencia se realiza de modo que cada paso es atómico, en tanto que solamente involucra a una construcción del lenguaje. Este tipo de semánticas presenta un nivel de abstracción más elevado que las semánticas operacionales basadas en máquinas abstractas. Trabajos que han seguido este enfoque, para lenguajes funcionales y funcionales paralelos, son [Lau93, BKHT99, BKT00].

Como de los sistemas reactivos interesa conocer todas las comunicaciones tangibles que los procesos han llevado a cabo, a partir de la semántica operacional se definen equivalencias entre procesos, como la bisimulación, o las pruebas, ligadas estas últimas también a las semánticas denotacionales. Un caso simple de equivalencia se tendría entre un proceso que implementara la ordenación de un *array* siguiendo el algoritmo *mergesort* y otro que lo hiciera según *quicksort*; desde el punto de vista del valor final que producen ambos procesos serían equivalentes.

3.1.2. Semánticas denotacionales

Este tipo de semánticas definen la función que computa un programa, pero no se ocupan de la forma en que dicha función se computa. Este mayor nivel de abstracción permite estudiar propiedades formales de los programas, como equivalencias entre estos. Constituyen interpretaciones matemáticas de los lenguajes. Una semántica denotacional consta de los siguientes elementos:

- Definición del espacio de significados: cada objeto sintáctico del lenguaje denota un elemento de este espacio. Cuando la sintaxis cuente con varias categorías semánticamente disjuntas se definirán diversos espacios. Los espacios pueden estar formados por funciones [Sto77], trazas [Hoa85], fallos [Hoa85], árboles de aceptaciones [DB97, Hen88, Fer96, FH99], etc.[1]
- Establecimiento, para cada constante del lenguaje, de su significado en el espacio definido a tal efecto, y construcción de funciones semánticas sobre el espacio de significados para cada operador del lenguaje.
- Definición de la función semántica principal que, empleando las anteriores, establece cuál es el valor semántico de cada programa. Se construye por inducción estructural sobre la definición de los términos del lenguaje.

[1] En realidad, conceptualmente puede tratarse de cualquier cosa, pues muchas veces no procede buscar interpretaciones naturales a los valores semánticos, o siendo ello posible podrían tener una gran complejidad (dominios definidos recursivamente, continuaciones, etc.).

Otro modo de definir semánticas denotacionales es por medio de las especificaciones que ha de verificar un programa, tal y como se hace en [OH86]. A partir de ellas se establecen propiedades de equidad, viveza, etc.

El nivel de descripción que estas semánticas emplean, independiente de cualquier implementación posible, las erigen como las ideales para el análisis de programas y el razonamiento sobre los mismos.

En ocasiones, estas semánticas se definen para que sean completamente abstractas (o equivalentes) a una equivalencia extraída de la semántica operacional, como pueden ser las *pruebas*, en las que un observador externo es quien determina la equivalencia o no entre los procesos. Para estas equivalencias se definen caracterizaciones alternativas a partir de las cuales se construyen las semánticas denotacionales. Las distinciones que se establezcan entre equivalencias de este tipo estarán íntimamente relacionadas con las capacidades que el observador externo posea. Relaciones de este tipo aparecen en [Hen88, Fer96, FH99] cuando se desarrollan las pruebas, o en [GMP90], estableciéndose equivalencias en términos de la bisimulación, y considerando un observador externo.

3.1.3. Semánticas axiomáticas

Este tipo de semánticas están definidas por medio de un conjunto de asertos o axiomas que definen las propiedades de un sistema e indican cuándo éstas son verificadas por la ejecución de un programa. La versión clásica [Hoa73] incluye pre-condiciones y post-condiciones que han de verificarse antes y después, respectivamente, de ejecutar un programa. Este tipo de semánticas se orienta principalmente a lenguajes en los que la ejecución del programa se basa en cambios de las variables de estado. Sin embargo, en los lenguajes funcionales las variables no son de este tipo y la aplicación de este tipo de semánticas a estos lenguajes no es adecuada. Esta línea de establecer axiomas como propiedades que tendrá que verificar la interpretación buscada del lenguaje ha sido empleada en el desarrollo de ACP (Álgebra de Procesos Comunicantes) en [BK84, Gla87]. En estas semánticas se trabaja con dos objetos, a saber, programas y fórmulas.

3.1.4. Semánticas algebraicas

Existe otro tipo de semánticas, las algebraicas, en las que solamente se consideran programas. Toda semántica define, en último término, una equivalencia entre procesos que quedaría definida directamente por medio de una axiomatización algebraica. Siguiendo esta filosofía, a partir de otra semántica ya definida, operacional o denotacional, se deducen propiedades y equivalencias entre procesos que constituyen un sistema de axiomas y se demuestra que dicho sistema de axiomas es correcto y completo con respecto a la semántica operacional o denotacional de la cual proviene. En este marco se sitúa la semántica presentada en [Hen88].

3.2. Semánticas formales de lenguajes funcionales paralelos y/o concurrentes

La concreción de las semánticas operacionales y denotacionales para lenguajes funcionales paralelos y concurrentes ha sido relativamente extensa. En las secciones que siguen se esbozan, sin dar ningún detalle formal, las características de algunas de estas semánticas, indicándose las referencias donde se puede encontrar su formalización. En [Hid99] se puede encontrar una recopilación de algunas de estas semánticas. En dicho trabajo se utiliza una sintaxis común, salvo particularidades propias de cada lenguaje, para la que se definen formalmente y con detalle las distintas semánticas. El lector interesado en los detalles técnicos puede recurrir a ese trabajo, en esta memoria solamente se incluyen comentarios informales, más o menos precisos, de las semánticas allí tratadas.

3.2.1. Semánticas operacionales

Las semánticas operacionales que vamos a considerar son las de dos dialectos concurrentes de ML, a saber, CML [Rep93] (CML1), [FH99] (CML2) y Facile [GMP90], y las de los dialectos paralelos de Haskell GpH [BKT00] y pH [AAA+95].

Las características que nos interesan de cada lenguaje, y que quedan recogidas en la Figura 3.1, son:

1. El tipo de evaluación que se sigue en las reducciones: *call-by-value*, *call-by-need* (ambas definidas en el glosario del Apéndice B), impaciencia, pereza[2] y *redex* (ver el glosario) siguiente a reducir.
2. Las construcciones sintácticas que definen los ámbitos de las variables.
3. En qué puntos el lenguaje introduce no-determinismo.
4. Cómo se introduce y trata la concurrencia o el paralelismo.
5. Cómo se llevan a cabo las comunicaciones.
6. Las equivalencias que se establecen entre los procesos.

3.2.1.1. Características de las semánticas operacionales para CML

Incluimos en esta sección las dos semánticas operacionales para CML mencionadas anteriormente. La primera de ellas es una semántica operacional de evaluación, en tanto que en el segundo caso se trata de una semántica operacional de cómputo.

[2] En pH , la evaluación en principio es perezosa, pues se basa en Haskell. Sin embargo, es impaciente por el paralelismo.

	Evaluación	Ámbitos variables	No-determinismo	Concur. Paralel.	Comunic.	Relaciones procesos
CML1	Impaciente *call-by-value* Izquierda a derecha	λ-abstracción `let` y `chan`	Elección y comunicaciones	Explícito	*Rendezvous*	
CML2	Impaciente *call-by-value*	`let` y `chan`	Elección y comunicaciones	Explícito	*Rendezvous*	Bisimulación y pruebas
Facile	Impaciente *call-by-value* Izquierda a derecha	λ-abstracción y `channel(t)`	Elección y comunicaciones	Explícito	*Rendezvous*	Bisimulación y observación
GpH	Perezosa *call-by-need* Más externo más izquierda	λ-abstracción y `let`	Orden de evaluación	Semi-explícito	Implícitas	Rendimiento y eficiencia
pH	Perezosa Estricta *call-by-need* *call-by-value* Más externo más izquierda	λ-abstracción y bloques	Implícito en celdas	Implícito $\Rightarrow$ impaciencia	Implícitas	

Figura 3.1: Características de los lenguajes

Características de la semántica de CML1

Esta semántica operacional, formalizada en [Rep93], está definida en dos niveles: la evaluación secuencial y la concurrente. La primera recoge la evaluación *call-by-value* y la evaluación de izquierda a derecha. La segunda refleja cómo se manejan las construcciones concurrentes: la generación dinámica de nuevos procesos y la comunicación entre ellos.

Se dota así a CML de una semántica cuyo principio de reducción queda patente en la gramática de contextos que se define en [Rep93]. Allí se exige que se evalúe en primer lugar la parte de la aplicación funcional correspondiente a la función, y en segundo lugar todos los argumentos, de izquierda a derecha, no procediéndose a la aplicación hasta que no haya finalizado la evaluación de dichos argumentos. El mismo esquema se sigue para la evaluación de la declaración local de variables.

Pasando a la parte concurrente del lenguaje, las configuraciones (pares formados por conjuntos de nombres de canal y por conjuntos de procesos) agrupan todos los procesos creados durante la ejecución del proceso inicial que da lugar a los procesos hijos, y los canales que cada uno de estos procesos crea para comunicarse con los demás. La relación de evaluación concurrente se define entre configuraciones, e indica cómo van evolucionando los procesos que se han ido creando.

La concurrencia propiamente dicha, es decir, la ejecución simultánea de varios procesos, queda plasmada en el hecho de que las configuraciones, en la relación de evaluación concurrente, mantienen conjuntos de varios procesos.

La evaluación se realiza sobre un único entorno, es decir, no se dispone de un concepto similar al de entorno local a cada proceso, y las comunicaciones se pueden realizar entre dos procesos cualesquiera en condiciones de sincronizar.

En cuanto al tipo de elección que se define en esta semántica, ésta introduce no-determinismo si tanto E_1 como E_2, en la expresión $E_1 + E_2$, pueden sincronizar con una tercera expresión. Sin embargo, el tipo de elección es determinista cuando solamente una de las dos expresiones está en condiciones de sincronizar.

El lenguaje incluye una construcción para sincronizar, que es una marca para llevar a cabo la comunicación de manera síncrona, de forma que, si no se aplica antes dicha construcción, la comunicación entre dos procesos no puede realizarse.

Además de los dos niveles de evaluación, se introducen trazas que reflejan de modo más fehaciente el no-determinismo que posee el lenguaje. En la evaluación concurrente se desarrolla sólo una de las múltiples reducciones posibles. En cambio, las trazas se recogen en un conjunto, de manera que cuando éste contiene más de un elemento es porque en un paso de reducción existe no-determinismo en una elección o comunicación, y cada traza del conjunto recoge cada una de las posibles opciones que en dicho paso se pueden tomar. Este concepto de trazas permite definir también cuando un proceso converge en una traza a un valor, y cuándo diverge: converge cuando en dicha traza existe una configuración que contiene al proceso ligado al valor, y diverge en caso contrario.

Por último, señalamos que la semántica que se presenta en [Rep93] no es composicional, pues la reducción de un término no se define en base a la reducción de las subexpresiones que lo componen.

Características de la semántica de CML2

Tanto en [Fer96] como en [FH99] se dota de semántica a un lenguaje con las características funcionales de ML y la inclusión de las comunicaciones y de la concurrencia en el estilo de CML, además de incluir un operador particular para evaluar en paralelo y construcciones propias de las álgebras de procesos.

Esta semántica operacional se construye tomando como base transiciones de los tipos siguientes: comunicaciones (envío y recepción de mensajes), actividad interna de un proceso y, lo que resulta más interesante, producción de valores y continuación tras la emisión de dichos valores. Con ellas se construyen tres grupos de *reglas*: las que rigen la producción de valores, las que son utilizadas en el proceso de reducción interna de una expresión, y las que regulan las comunicaciones.

El principal objetivo de esta semántica es la *producción de valores*. Sin embargo, con transiciones que sólo recojan dicha producción y no expresen lo que un proceso puede

realizar posteriormente, no se abarca toda la capacidad que se pretende que tenga el lenguaje, pues se perderían todas las comunicaciones de las que sería capaz la continuación del proceso en cuestión con el resto de procesos vivos. Además, la producción de un valor se puede llevar a cabo a lo sumo en una ocasión durante la evaluación completa de un proceso.

En cuanto a la parte puramente funcional del lenguaje, al tratarse de una extensión de ML, se desea que se mantenga la estrictez. Esta propiedad queda reflejada en la regla de reducción correspondiente a la construcción `let`, en la que se exige como premisa que se haya evaluado la expresión ligada a la variable local para proceder con la evaluación de la expresión que es el ámbito de dicha variable.

La parte concurrente del lenguaje presenta no-determinismo en la elección definida a tal efecto, en la que, por medio de una acción interna, se puede efectuar la elección de modo arbitrario entre las dos expresiones involucradas; la comunicación entre procesos también conduce a no-determinismo, pues si varios procesos pueden emitir valores por un canal, no hay regla que favorezca el acceso de unos frente a otros, y si varios se encuentran en disposición de recibir valores por un canal, el que comunica elegirá arbitrariamente a cuál se lo envía. En cualquier caso, la comunicación en este lenguaje es por medio de *rendezvous*, como sucedía en [Rep92, Rep93, PR97].

Para dar semántica a la creación dinámica de procesos se empleará un operador de paralelo, de manera que el proceso nuevo se ejecutará en paralelo con los anteriormente generados. Se explica así la coexistencia en el lenguaje de un operador generador de procesos nuevos y otro de evaluación en paralelo, pues el segundo no es más que una herramienta auxiliar para dotar de semántica al primero.

Además, el paralelo se emplea para establecer la semántica de otras construcciones del lenguaje, como el `let`, de modo que se permite que una expresión pueda ejecutarse como un proceso en paralelo con los restantes si ya ha producido su valor. Así se permite que se produzcan valores adicionales por canales de comunicación mientras continúa la interacción con el resto del programa.

Con esta semántica operacional, es posible la definición de equivalencias entre procesos en términos de bisimulaciones. Con ellas, se puede "eliminar" el operador de creación de procesos, que queda expresado en función del operador de paralelo; y se establecen equivalencias entre expresiones `let` que recogen las principales características de las λ-abstracciones: la abstracción identidad, la β-reducción y la realización de un cierto tipo de currificación.

A diferencia de la semántica de [Rep93], ésta sí que es composicional, pues la semántica de una expresión viene determinada por la de las subexpresiones más simples que la conforman.

3.2.1.2. Características de la semántica de Facile

En esta semántica [GMP90], y a diferencia de las descritas brevemente en las secciones anteriores, se tiene presente en todo momento el tipo de cada parte del programa, es decir, en la semántica dinámica se considera la semántica estática.

La semántica dinámica se basa en dos relaciones de transición, una que regula el comportamiento de las expresiones ordinarias (relación de evaluación) y otra que establece cómo han de ser reducidas las expresiones de comportamiento (relación de derivación).

En cuanto a las características concurrentes del lenguaje, la comunicación que plantea esta semántica, al igual que en las presentadas en [Rep93, FH99], es de naturaleza síncrona, hecho que queda patente cuando se definen las acciones de comunicación y se les obliga a complementarse a la hora de comunicarse dos procesos. Por otro lado, el operador de elección es no-determinista y se permite que la realización de una acción de cualquier tipo (de comunicación, interna o de expresión de comportamiento) ejecute la elección; esto hace que la elección que aquí se propone sea distinta de la contemplada en la semántica de CML definida en [FH99], si bien esta última disponía de dos operadores de elección: interna y externa.

Si se observa el programa como un todo, su comportamiento viene determinado, además de por la semántica operacional, por una observación de carácter externo basada en la comunicación: se definen ventanas, o conjuntos de canales visibles para un observador externo, las cuales ocultan canales locales; además, a causa de la movilidad del lenguaje, dichas ventanas no tienen un carácter estático, pues se pueden hacer visibles canales que antes eran locales, que entonces se incorporarán a la ventana del observador. Las ventanas son, por tanto, un método de observar procesos que varían su interfaz mientras se están ejecutando. Partiendo de una ventana inicial, y empleando la observación externa, se define la equivalencia entre procesos.

Este modo de ver la equivalencia entre procesos recoge tanto la esencia de la bisimulación, en tanto que dos procesos para ser equivalentes han de poder simularse mutuamente, como la de la semántica de pruebas, pues se exige la participación de un observador externo, aunque éste no aparezca en forma de test y sólo permanezca de él lo que puede ver (la ventana de canales).

Esta semántica, como indican sus reglas operacionales, es composicional.

3.2.1.3. Características de la semántica de GpH

La semántica de [BKHT99, BKT00], que será detallada en el Capítulo 5 por ser la base de este trabajo, presenta una diferencia fundamental con respecto a las anteriormente perfiladas: es una semántica de más bajo nivel o más cercana al funcionamiento de una máquina en la que se evalúan los programas, hecho que queda patente al utilizar *heaps* para recoger las ligaduras de ejecución y etiquetar éstas con los estados por los que

pasan. No es, por tanto, una semántica que se limite a indicar el proceso de reducción de un programa, pues se basa en el modelo de ejecución subyacente para GpH.

Se trata de una extensión de la semántica natural de Launchbury [Lau93], también detallada en el Capítulo 5, para incluir las dos primitivas de paralelismo y secuencialidad: `par` y `seq`. Se demuestra que la extendida de [BKHT99] es correcta con respecto a la de [Lau93] si se cuenta con un único procesador; además, ambas son equivalentes si a la natural de Launchbury se le añaden las dos primitivas `par` y `seq`.

En cuanto a las características funcionales del lenguaje, la semántica presta especial atención a la pereza, quedando recogida en el hecho de incorporar las clausuras relativas a las variables locales y a los argumentos de las λ-abstracciones en el *heap* y retrasar su evaluación (marcándolas como ligaduras inactivas) hasta que otra variable demande, para su evaluación, el valor que tienen ligado.

Al disponerse de un *heap* de ligaduras etiquetadas es posible que entre ellas puedan compartir información. Por ejemplo, en el caso de que una variable sea utilizada por otras dos, si una de éstas provoca su evaluación, cuando la otra la necesite para realizar la suya ya estará evaluada.

La parte paralela del lenguaje se refleja en el hecho de contar con varias hebras de ejecución (asimilables a los procesos de los lenguajes descritos con anterioridad) que pueden ser ejecutadas al mismo tiempo, de modo que si el sistema dispone de procesadores libres pueden estar ejecutándose simultáneamente, pasando así a ser hebras activas. La generación de ligaduras se realiza mediante la incorporación de las clausuras de un `let` al *heap* (todas las clausuras se incorporan a la vez, pues la declaración local de variables es recursiva). Con ayuda del operador `par` se da oportunidad al programador de indicar los puntos en los que es posible ejecutar el programa en paralelo.

La semántica se define por medio de dos tipos de transiciones: las de paso simple y las de paso múltiple. Con las primeras se establece cuándo una hebra puede evolucionar en un paso de evaluación y cómo dicha evaluación afecta al resto de las hebras del *heap*; sin embargo, con este tipo de transiciones no se pueden tener en el *heap* varias hebras activas a la vez, por lo que se hace necesario otro conjunto de transiciones, las de paso múltiple, que tienen dos tareas fundamentales: ejecutar varias hebras en paralelo y planificar los procesadores.

Estos dos niveles de evaluación se asemejan a los que se presentan en la semántica de [Rep93]. Sin embargo, en tanto que en aquella semántica para CML bastaba con realizar la evaluación secuencial cuando no se deseaba paralelismo, en la de GpH es imprescindible el uso de las transiciones de paso múltiple con un único procesador para llevar a cabo la evaluación secuencial de un programa, en otro caso, si la hebra activa pasara a bloquearse o quedara inactiva, no sería posible activar otra hebra distinta.

A pesar del no-determinismo introducido por la posibilidad de activar distintas hebras en un mismo paso de planificación, el lenguaje pierde la propiedad de confluencia, y aunque se puedan realizar distintas reducciones, el valor obtenido por cada una de

ellas como resultado del programa es el mismo.

Las semánticas operacionales que definiremos para nuestro lenguaje tienen por cimiento ésta que ahora se acaba de esbozar, y que en el Capítulo 5 se detallará formalmente.

3.2.1.4. Características de la semántica de pH

Primero señalar que, a diferencia de GpH, también basado en Haskell, pH incluye dos tipos de aplicación: perezosa y estricta. En consecuencia, los modelos semánticos que se definan para pH tendrán que conjugar los dos tipos de evaluación.

La semántica para pH presentada en [AAA+95] describe el comportamiento de las expresiones del lenguaje en base a la evolución de una máquina abstracta que lleva el alma de la máquina-G [Pey87]. Su estructura está conformada por varios procesadores, todos ellos compartiendo una única memoria, y a los que cuando están ociosos se les alimenta de trabajo a partir de una cola establecida a tal efecto. Una descripción gráfica de esta máquina se incluye en la Figura 3.2.

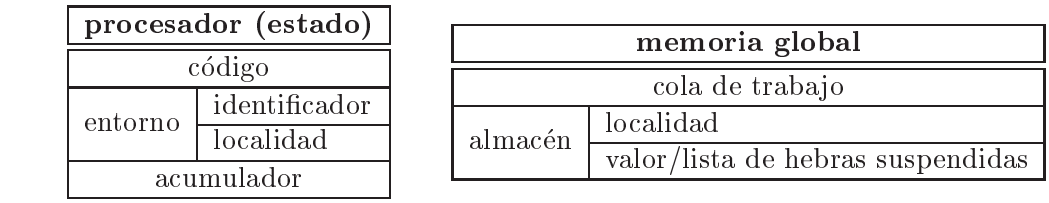


Figura 3.2: pH: máquina abstracta

La instrucción principal de la máquina es *eval*, que interpreta en un entorno una expresión por medio de su descomposición estructural. En cada paso, el procesador ejecuta la instrucción que se encuentra en la cabeza de su secuencia de código, modificando el estado del procesador. La cola de trabajo es una estructura FIFO (primero en entrar primero en salir) para garantizar la justicia del planificador. El estado global de la máquina lo componen los estados individuales de cada proceso junto con la memoria global. La máquina comienza su ejecución planificando, es decir, llevando a un procesador una secuencia inicial de instrucciones para evaluar el programa; a continuación imprime el resultado del acumulador; y luego continúa planificando el resto del trabajo que aún permanezca en la cola. La instrucción de impresión solamente va asociada con la hebra principal de ejecución, y el hecho de que pueda imprimirse antes de terminar la ejecución abunda en la naturaleza no estricta del modelo de evaluación (no estricta con la salvedad de la aplicación estricta). La finalización se produce cuando tanto la secuencia de código como la cola de trabajo están vacías.

Además de la instrucción principal, la máquina tiene otras instrucciones que son invocadas para evaluar cada construcción del lenguaje. Especial atención merecen `touch`,

que comprueba si en una localidad hay o no un valor, permitiéndo modelizar ambos tipos de aplicación y la estrictez, y las empleadas para implementar las operaciones sobre celdas (junto con sus efectos laterales) `alloc`, `fetch` y `store`.

El paralelismo queda modelizado por el hecho de tener varios procesadores e ir distribuyendo entre ellos las secuencias de código de la cola de trabajo. Cada expresión asociada a una variable local dará lugar a una de estas secuencias.

En cuanto al no-determinismo, éste viene también refrendado por este reparto y por el hecho de que cuando se almacena un valor en una localidad, las hebras suspendidas en esta localidad pasan a la cola de trabajo.

El significado de pH se ha definido también por medio de una semántica operacional alternativa en [NA01]. Esta semántica se caracteriza por ser composicional del tipo que Plotkin describe en [Plo81]; se define el significado de pH en base a una serie de reglas que reescriben las distintas expresiones sintácticas del lenguaje.

3.2.2. Semánticas denotacionales para CML

Al igual que sucedía en el caso de las semánticas operacionales, vamos a considerar dos semánticas denotacionales para CML, lenguaje que se ha mostrado muy fructífero a la hora de dotarle de significado formal. La primera de ellas, proveniente de [DB97], está basada en la semántica estática de tipos sobre la que se construyen árboles de aceptaciones, ya empleados para tratar la semántica formal de álgebras de procesos en [Hen88]. La segunda, presentada en [FH99], tiene también por dominio semántico los árboles de aceptaciones, pero en esta ocasión no son construidos a partir de la semántica estática, sino que se definen a partir de un conjunto de eventos o acciones que incluyen el envío y la recepción de mensajes y una acción especial que indica la terminación inmediata y con éxito del proceso.

3.2.2.1. Características de la semántica basada en tipos

Esta semántica, presentada en [DB97], fundamenta su modelo en los árboles de aceptaciones, teniendo por acciones las extraídas de las comunicaciones por canales y la que denota la devolución de un valor. En ella, cada proceso se define por medio de acciones y funciones de continuación, como en el modelo general, y además tiene asociado un conjunto, su tipo, que define los posibles resultados que el proceso puede emitir. A diferencia de la semántica de [FH99], después de la producción de un valor, el proceso no tiene continuación posible. Aunque no se exponga ninguna propuesta de semántica basada en la observación externa, el modelo de árboles de aceptaciones que desarrolla esta semántica hace que el significado de un proceso sea aquello que desde el exterior puede apreciarse, pues las únicas acciones posibles son de comunicación.

La construcción de esta semántica procede definiendo el dominio semántico y, a con-

tinuación, la función semántica principal, junto con las auxiliares que se necesitan para su descripción. El dominio se fundamenta sobre una serie de espacios estáticos (tipos), cada uno con su contrapartida dinámica. A partir de estos dominios dinámicos se construye el dominio de procesos sobre el que se evaluarán semánticamente las expresiones. Al ser el sistema de tipos la base de esta semántica denotacional, se dice que la misma está tipada por la semántica estática.

La función semántica principal se define mediante una serie de reglas que, teniendo en cuenta la semántica estática de una expresión —inferida de manera composicional— determinan la semántica dinámica o denotacional de la expresión. La información estática no incluye solamente el tipo de una expresión, sino también todos aquellos efectos "laterales" producidos por la creación de canales y la comunicación entre procesos.

Para ello se utilizan funciones auxiliares que definen la semántica de las constantes y de los operadores. En ellas aparecen los puntos de no-determinismo en el lenguaje: la elección interna, la elección externa, que se tiene cuando ambos procesos pueden realizar la misma acción, y el paralelo, en tanto que se pueda elegir entre la evolución independiente de uno u otro proceso y la conjunta.

La definición de la semántica estática de cada expresión es composicional, pero la dinámica no lo es, pues no emplea la semántica de cada subexpresión, sino los tipos y efectos de la expresión completa. Además, la construcción del dominio dinámico tampoco es composicional, pues el polimorfismo se define por medio de tipos dependientes.

3.2.2.2. Características de la semántica de árboles de aceptaciones

Se construye en [FH99] un modelo denotacional con el que se pretende capturar la producción de valores que presenta CML.

La definición toma como base teórica los árboles de aceptaciones [Hen88] y, por lo tanto, un proceso semántico vendrá definido por las acciones que puede realizar y por la forma en que continúa su evolución tras ejecutarlas. Las acciones serán de comunicación y de producción de valores; en definitiva son aquéllas que un observador externo puede ver. Este observador sería equivalente a un *test* de los que definen el preorden estudiado en [FH99], pues las acciones de los conjuntos de aceptaciones son las mismas que las que aparecen en las pruebas; por esta razón se introduce en esta semántica denotacional el preorden de pruebas extraído de la semántica operacional de producción de valores.

El resto de funciones necesarias son las que dotan de semántica a cada operador del lenguaje. Dentro de los operadores no se incluye la creación dinámica de procesos, `spawn`, porque, al igual que se hace en algunas semánticas operacionales [FH99, GMP90], su semántica se define tomando como base la del operador paralelo.

Es en algunas de estas funciones donde se introduce el no-determinismo, concretamente en las referentes a los operadores de elección y paralelo. La elección interna es no-determinista por naturaleza, en tanto que la externa presenta esta característica

cuando una acción está presente en sendos elementos de los conjuntos de aceptaciones de los dos procesos. Por su parte, el operador paralelo es no-determinista cuando se da la posibilidad de evolución conjunta e independiente a la vez, siendo entonces la elección interna la que escoge entre todas las posibles evoluciones, o cuando ambos procesos pueden evolucionar independientemente, teniendo entonces que elegir cuál de ellos lo hace en primer lugar.

Esta semántica, al ser denotacional, se caracteriza por su composicionalidad, patente en la función semántica principal.

Terminamos aquí el paseo por semánticas definidas para lenguajes funcionales concurrentes y paralelos. En el camino operacional nos hemos detenido en aquellas que presentan más interés de cara a la definición de semánticas formales para Eden. Por ejemplo, será la semántica de GpH la que extenderemos y adaptaremos para Eden. En cuanto a las denotacionales, se han descrito informalmente dos semánticas para un mismo lenguaje, que construyen el significado de las expresiones del lenguaje del mismo modo, aunque el árbol final se compone a partir de distintos elementos: tipos de la semántica estática en la primera y acciones en la segunda.

CAPÍTULO 4

El lenguaje Jauja

> Los límites de mi lenguaje son los límites de mi mundo.
>
> Ludwig Wittgenstein

En nuestra opinión, este trabajo *no es jauja*, pero sí que es cierto que será en Jauja donde se apoyen nuestras lucubraciones, pues Jauja va a ser el lenguaje objeto de nuestras semánticas. Sin embargo, Jauja no surge de la nada, sino que proviene del lenguaje funcional paralelo Eden [BLOP96a, BLOP96b, BLOP97]. Tampoco el nombre elegido es arbitrario: el principio de este trabajo se sitúa en *el Edén*, nombre del lenguaje que es el *paraíso del paradigma funcional paralelo* [BLOP96a]. Otros desarrollos vinculados a Eden han adquirido nombres relacionados: Paradise (PARAllel DIstribution Simulator for Eden) [HPR00, Rub01], DREAM (the DistRibuted Eden Abstract Machine) [BKL+97], el sueño en el que los deseos formulados en Eden son ejecutados. Y siguiendo esta dinámica elegimos Jauja, que según [Mol96] se aplica a "un sitio o situación en que hay bienestar y abundancia", que según [Rea95] es la voz que se utiliza "para denotar todo lo que quiere presentarse como tipo de prosperidad y abundancia", y que la leyenda atribuye como nombre a un fabuloso país.[1] Situémonos entonces en el origen y adentrémonos en el huerto delicioso.[2]

[1] El origen de la leyenda del fabuloso país de Jauja está en los primeros relatos de Pizarro, escritos en Hatun Xauxa, en la región peruana de los huanca. Se identificó el nombre de esta ciudad con todas las supuestas riquezas de Perú. La leyenda se difundió también por Italia y Francia. Entiéndase ahora la figura que se incluía al principio de esta memoria.

[2] *Éden*: origen hebreo del vocablo edén, cuyo significado, en dicha lengua, es "huerto delicioso".

4.1. Entrando en *el Edén*

Siendo un hecho que paralelismo y distribución constituyen mejoras en lo referente a la eficiencia de la programación, la intención de los creadores de Eden era sacar beneficio de ambos en el ámbito del paradigma funcional. Los lenguajes funcionales se basan en el λ-cálculo ideado por Alonzo Church [Chu41]. Así sucede con Haskell [PH99, Pey03], el lenguaje funcional perezoso por excelencia, que además es polimórfico en sus tipos. El lenguaje Eden tuvo sus bases en CFP [BLO94] y se concretó en [BLO96, BLOP96b, BLOP96a, BLOP97]. Eden extiende Haskell con construcciones para definir procesos de manera explícita. El programador en Eden dispone de control absoluto sobre la granularidad de los procesos, la distribución de los datos y la topología de comunicaciones, pero desde un elevado grado de abstracción, hecho que viene refrendado por no tener que preocuparse de tareas de sincronización.

Las características fundamentales de Eden se resumen en:

Abstracciones de proceso: son las expresiones que de un modo puramente funcional definen el comportamiento general de un proceso.

Creaciones de proceso: son aplicaciones de las anteriores a un grupo determinado de expresiones que conformarán los valores de los canales de entrada del nuevo proceso creado.

Comunicaciones entre procesos: son asíncronas e implícitas, pues el paso de mensajes no lo ha de explicitar el programador. Además, estas comunicaciones no tienen por qué ser de un único valor, sino que pueden conformarse en forma de *streams*.

Sin embargo, las maravillas *del* Eden se extienden hasta incluir construcciones para modelizar sistemas reactivos:

Creación dinámica de canales: sin esta facilidad las comunicaciones son solamente jerárquicas entre el proceso padre y el proceso hijo. Los canales dinámicos permiten romper esta jerarquía, pudiéndose generar así topologías de comunicación más complejas [PRS02].

No-determinismo: con el objetivo de modelizar las comunicaciones de varios a uno se introduce la abstracción de proceso predefinida `merge`, que toma varios *streams* y devuelve uno único, compuesto por una mezcla no-determinista de los elementos de los anteriores.

De lo expuesto hasta ahora se deduce que Eden está compuesto por dos capas:

1. Nivel de funciones, o modelo computacional.
2. Nivel de procesos, o modelo de coordinación.

Detengámonos en la coordinación. En primer lugar, los procesos son entes aislados salvo por la existencia de comunicación entre ellos, que se realiza por medio de canales.

Así, cada proceso dispone de su memoria particular y si un proceso necesita un valor que tiene otro, solamente podrá disponer de él si éste se lo comunica a través de un canal. En el caso de que el canal sea un *stream*, o lista, su cabeza será estricta, de modo que ésta se evaluará dando lugar a un valor comunicable. Sin embargo, la evaluación de la lista completa será perezosa, introduciendo de este modo la posibilidad de tener canales que definan *streams* potencialmente infinitos.

El no-determinismo permite que Eden deje de ser puramente transformacional para pasar a ser reactivo. La forma de introducirlo mezclando varios *streams* en uno único permite que el proceso que genera el no-determinismo no tenga que estar esperando a que un *stream* concreto le envíe un valor, sino que cualquiera de los conectados puede ir enviando los suyos y el receptor los irá incluyendo a la lista final en el orden en que lleguen. La relación entre este comportamiento y la reactividad es ampliamente tratada en [Hen82].

A pesar de tener un núcleo perezoso, la introducción de paralelismo provoca una pérdida de pereza en la evaluación de determinadas expresiones:

- La comunicación no se realiza bajo demanda. Por el contrario, cuando se crea un proceso, automáticamente se da vía libre a la evaluación de las expresiones que darán lugar a los valores que vayan a ser comunicados por sus canales.
- La creación de procesos no se retrasa hasta que la salida producida por el nuevo proceso sea demandada, sino que en el momento en el que la expresión de creación es localizada, lo que sucederá al evaluar una declaración local en la que una de sus variables se encontrará ligada a dicha expresión de creación, se procederá a la creación del nuevo proceso.

Por supuesto que la evaluación sigue siendo no-estricta en lo referente a la aplicación funcional. Pero, además, en el caso de que un proceso no necesite el valor del canal de entrada, la evaluación de dicho proceso de desarrollará con normalidad aunque la evaluación del canal no dé lugar de momento a ningún valor.

Por otra parte, la evaluación de una expresión termina cuando ésta se encuentra en forma normal débil de cabeza (*whnf*) (su definición puede consultarse en el Apéndice B), salvo cuando se trata de comunicar un valor, pues si éste no fuera una λ-abstracción habrá de evaluarse hasta forma normal (ver Apéndice B).

4.2. Jauja

El lenguaje Jauja, del que vamos a definir significado vía semánticas formales en este trabajo, es una simplificación de Eden en la que se incluirán construcciones sintácticas que satisfarán los mínimos arriba indicados. Así, Jauja también está compuesto por dos partes diferenciadas:

- λ-cálculo perezoso: es la parte puramente funcional de Jauja.

- Expresiones de coordinación: permiten introducir paralelismo mediante la creación explícita de procesos, que interaccionan usando canales de comunicación, e incorporar no-determinismo, y por tanto reactividad, en Jauja.

El λ-cálculo sin tipos extendido se detalla en la Figura 4.1. Además de la sintaxis recogida en esta figura, en los ejemplos aparecerán números naturales como azúcar sintáctico de las λ-expresiones que convenimos que los representan. Por ejemplo 4 es una abreviatura de la expresión $\backslash f.\backslash x.f(f(f(f\,x)))$. De igual manera incluimos operadores aritméticos.

Jauja			
E	$::=$	x	variable
	$\mid$	$\backslash x.E$	λ-abstracción
	$\mid$	$E_1 E_2$	aplicación
	$\mid$	$E_1 \# E_2$	creación de procesos
	$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	declaración local
	$\mid$	$\texttt{new}(y, x)E$	canal dinámico
	$\mid$	$x\,!\,E_1\ \texttt{par}\ E_2$	conexión dinámica
	$\mid$	$E_1 \bowtie E_2$	mezcla
	$\mid$	$\Lambda[x_1 : x_2].E_1 [\![E_2$	Λ-abstracción
	$\mid$	L	lista
L	$::=$	$\texttt{nil}$	lista vacía
	$\mid$	$[E_1 : E_2]$	lista no vacía

Figura 4.1: Jauja: sintaxis

En primer lugar, se dispone de *variables*, que posteriormente facilitarán que las expresiones sean compartidas. Denotaremos el conjunto de todas las variables por Var.

Fundamentales en un λ-cálculo son la *abstracción funcional* y la *aplicación*. La aplicación especial # desempeña el papel de *creadora de procesos*. Desde el punto de vista de la obtención del valor final al que da lugar una expresión, la creación de procesos es equivalente a la aplicación funcional. No obstante, si se observan operacionalmente los procesos, los cambios que se producen en el sistema de ejecución son cualitativamente distintos:

- La evaluación del argumento de una aplicación es perezosa, en tanto que la de una creación de proceso (canal de entrada al nuevo proceso) es impaciente, con lo que en el segundo caso pueden crearse procesos que en el primero no existirían:

  Ejemplo 4.1 *Efectos laterales en la evaluación de los argumentos de las aplicaciones.*

  $$E_1 \equiv (\backslash x.(3))((\backslash x.x)\#1)$$
  $$E_2 \equiv (\backslash x.(3))\#((\backslash x.x)\#1)$$

  En tanto que en la primera expresión el proceso $((\backslash x.x)\#1)$ no se crea, en la segunda sí, debido a la evaluación impaciente del argumento.

  □

- En el caso de la aplicación, las variables libres de la abstracción solamente se evalúan bajo demanda. Sin embargo, en el caso de la creación de procesos, dependiendo de la opción de distribución de datos entre procesos que se tome, caben dos posibilidades: evaluación sólo en caso de necesidad, o evaluación previa para generar el entorno inicial del nuevo proceso. Ambas opciones serán tratadas y descritas formalmente en este trabajo.

El efecto de evaluar una expresión del tipo $E_1 \# E_2$ es la creación de un nuevo proceso y de dos canales de comunicación, a saber, uno de entrada al hijo, por el que el padre le comunicará el resultado de la evaluación de E_2, y otro de salida del hijo, por el que éste devuelve al padre el valor correspondiente a la aplicación $E_1\ E_2$ (ver Figura 4.2).

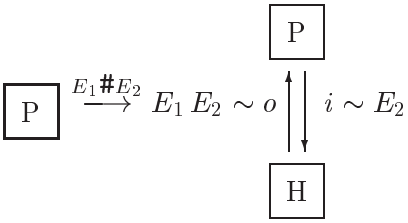


Figura 4.2: Esquema de creación de procesos

La distribución de datos entre los procesos tiene como principio fundamental que cada proceso sea una entidad independiente, exceptuando la comunicación mediante canales. Para conseguir este nivel de aislamiento es necesario que cada proceso tenga almacenada en su memoria particular la información que necesita para evaluar su salida principal. Así, cuando se crea un proceso se le asocia su entorno inicial de memoria que incluye todas las variables libres de la abstracción. Es en este punto donde caben las dos posibilidades mencionadas anteriormente:

1. que todas las variables libres tengan asociados valores ya evaluados,
2. que, por el contrario, estén ligadas a expresiones posiblemente sin evaluar.

La segunda de las opciones da lugar a una potencial duplicación de la evaluación de una misma expresión, en tanto que la primera solamente evalúa cada variable una única vez.

Ejemplo 4.2 *Entorno inicial en la creación de un proceso.*

$$\begin{aligned} &\texttt{let}\ x = \backslash t.y, \\ &\quad\ \ y = (\backslash s.s)3, \\ &\quad\ \ z = 4 \\ &\texttt{in}\ x\#z \end{aligned}$$

La única variable libre del proceso $x\#z$ es y. Caben dos posibilidades para generar el entorno inicial de este proceso: que el mismo contenga las variables x e y ligadas a las

expresiones tal como se encuentran en este momento, o bien evaluar la variable y hasta obtener el valor 3 para copiarlo sólo entonces.

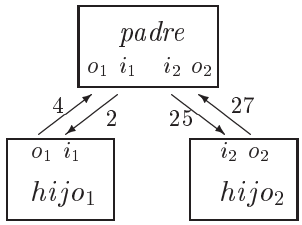
□

Ante la presencia de no-determinismo, la decisión anterior no es trivial, pues en el segundo caso una misma variable podría dar lugar a valores distintos en cada ocasión en que se evalúe.

La *declaración local* de variables, además de cubrir sus funciones habituales, es el mecanismo que permite que tengamos la ejecución en paralelo de varios procesos. La evaluación perezosa de una expresión no conlleva paralelismo alguno. Sin embargo, cuando se permite que determinadas expresiones de una declaración local se evalúen al tiempo que su cuerpo, se tiene evaluación paralela. La explotación en mayor medida del paralelismo explícito, introducido mediante `#`, se realiza cuando una de las variables locales está ligada a una creación de proceso (*nivel superior*), pues éste será creado sin necesidad de ser demandado por ninguna otra expresión. De modo que, cuando nos encontremos en esta situación, los procesos se crearán de manera especulativa. En el Ejemplo 4.3 se observan las dos vías que conducen a la creación de procesos.

Ejemplo 4.3 *Creaciones de procesos*

```
let
  x = y#25, y = \x.x + 2
in y#2
```



Esta expresión da lugar a la creación de dos procesos: la del primero de ellos se realiza bajo demanda ($hijo_1 \equiv y\#2$), en tanto que la del otro, al tratarse de una `#`-expresión de nivel superior, se realiza de manera especulativa ($hijo_2 \equiv y\#25$). Dependiendo de las condiciones de evaluación que se estén considerando, el segundo proceso llegará o no a producir su valor.

□

Sin embargo, la comunicación jerárquica padre-hijo heredada de Eden no es todo lo eficiente que se desearía. Por ejemplo, la comunicación directa entre dos hermanos es más eficiente que si ha de realizarse a través del padre. Por ello se incluyó en Eden, y también en Jauja, una construcción para crear *canales dinámicos*. Dichos canales son creados en dos tiempos (ver esquema en la Figura 4.3): la primera parte de la creación, `new`$(y, x)E$, la lleva a cabo el proceso consumidor C que recibirá valores a través del nuevo canal, para lo cual envía el nombre asociado, y, al que será el nuevo canal. Dicho nombre será enviado a través de una serie de procesos, $P_1, \ldots, P_n$ $(n \geq 0)$, hasta llegar al proceso P que lo usa. En dicho proceso la evaluación de la expresión $(y\,!\,E_1$ `par` $E_2)$ conlleva la evaluación en paralelo de E_1 y de E_2, siendo este paralelismo de hebras,

como el descrito en la Sección 3.2.1.3 para GPH, y no de procesos. El valor resultante de evaluar E_1 se envía a x en el proceso consumidor, C, donde será utilizado en el ámbito de la expresión E.

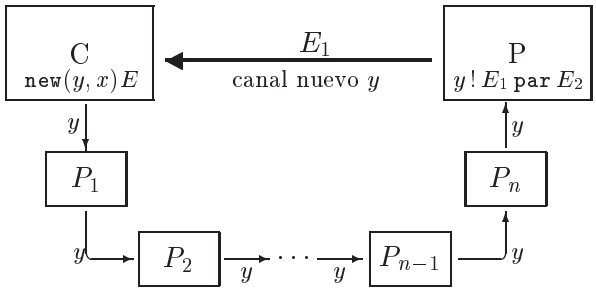


Figura 4.3: Esquema de canales dinámicos

Nótese que no sólo P puede establecer el canal dinámico con C, sino que cualquiera de los procesos —en la Figura 4.3 P_i— que posean el nombre del canal puede establecer dicha conexión. De este hecho se deriva la aparición de no-determinismo implícito en Jauja. Eso sí, solamente es posible una conexión con ese nombre de canal. En caso de intentar una nueva conexión se produce un error en tiempo de ejecución.

Sea cual sea el tipo de comunicación en juego, ésta siempre es asíncrona, de modo que no existen primitivas de comunicación del tipo `send` o `receive`, como las que aparecen en CML [Rep93] o en el π-cálculo [SW01].

Aparte del no-determinismo implícito derivado de la existencia de canales dinámicos, se incluye en Jauja el *no-determinismo* explícito de Eden del tipo de la elección interna de las álgebras de procesos [Hen88, Hoa85, Mil89]. El operador $\bowtie$ toma como argumentos dos expresiones; cada una de ellas dará lugar a un *stream*, siendo ambos posteriormente mezclados en uno solo; según cada uno de ellos vaya teniendo en la cabeza un valor comunicable, éste pasará a formar parte del *stream* final, presentándose el no-determinismo cuando ambos *streams* tienen en cabeza un valor comunicable. La mezcla se dará por concluida cuando los dos *streams* hayan terminado.

Para tratar con *listas*, y por ende con *streams*, se introduce una forma especial de abstracción $(\Lambda[x_1 : x_2].E_1 \| E_2)$ que encaja patrones:

- Lista vacía: la evaluación continúa con E_1.
- Lista no vacía: se procede con la evaluación de E_2, tras las correspondientes sustituciones de parámetros formales por argumentos actuales.

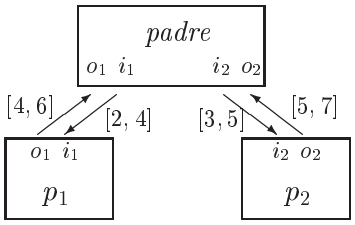
Esta necesidad de distinción entre lista vacía y no vacía provoca que esta aplicación sea estricta en su argumento. En consecuencia, estamos ante la presencia de tres tipos de aplicación: perezosa (funcional), impaciente (creación de procesos), y estricta (listas).

Finalmente, se introduce la categoría sintáctica de las listas, que, como se deduce de lo anterior, incluye la lista vacía, `nil` y la no vacía. La evaluación de una lista es perezosa, considerándose evaluada cuando se obtiene el constructor en cabeza. Sin embargo, como se mencionó anteriormente, cabe la posibilidad de evaluar las listas perezosamente pero con demanda de cabeza, tal es el caso de los *streams* de comunicación.

Veamos por medio de ejemplos la conjunción de listas y no-determinismo en Jauja:

Ejemplo 4.4 *No-determinismo.*

```
let
   x = z#[2 : [4 : nil]],
   y = z#[3 : [5 : nil]],
   z = Λ[x1 : x2].nil‖[x1 + 2 : z x2]
 in x ⋈ y
```



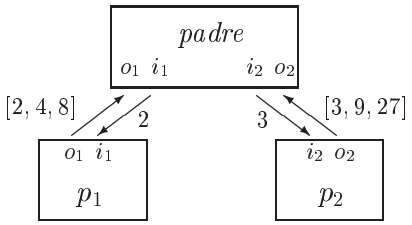
La creación de los procesos de este programa, p_1 y p_2, lleva a la evaluación de sendos *streams* que son los argumentos de $\bowtie$. Los resultados posibles del programa son: $[4, 6, 5, 7]$, $[4, 5, 6, 7]$, $[4, 5, 7, 6]$, $[5, 4, 6, 7]$, $[5, 4, 7, 6]$ y $[5, 7, 4, 6]$.

□

No obstante, un programa que en su código incluya el operador no-determinista no conduce necesariamente a no-determinismo en la producción de valores, como se muestra en el siguiente ejemplo.

Ejemplo 4.5 *No-determinismo determinista.*

```
let s = Λ[y1 : y2].(0)‖(y1 + s y2)
in s (let p = \x.[x : [(x ∗ x) : [(x ∗ x ∗ x) : nil]]]
        in (p#2) ⋈ (p#3))
```



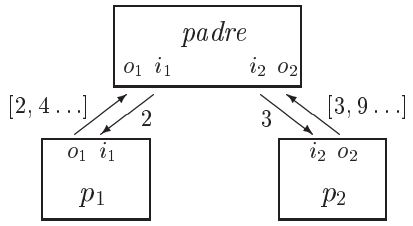
Como el resultado de este programa es la suma de todos los elementos de un *stream*, el valor resultante es independiente del orden de los elementos de dicho *stream*.

□

También es posible la creación de *streams* infinitos:

Ejemplo 4.6 *Streams infinitos.*

$$\begin{array}{l}\texttt{let } s = \lambda z.\Lambda[y_1 : y_2].z[\![(s\,(z + y_1)\,y_2)\\ \texttt{in } s\,0\,(\texttt{let } p = \backslash x.[x : p(x * x)]\\ \qquad \texttt{in } (p\,\#\,2) \bowtie (p\,\#\,3))\end{array}$$



El valor de salida del proceso *padre*, es decir, la suma de los dos *streams* que producen los hijos, nunca se producirá pues la mezcla sólo se construye parcialmente.

□

4.3. Normalización

Antes de definir el significado de una expresión de Jauja, se procede a un proceso habitual de normalización [AAA+95, BKT00, HO02b, Lau93] en el que se introducen variables y declaraciones locales, de manera que todas las aplicaciones contengan variables tanto en la parte de la abstracción como en la del argumento, y que la mezcla no-determinista tenga por argumentos dos variables. Asimismo, el cuerpo de una declaración local será una variable, por supuesto incluida dentro del conjunto de variables locales, para que la expresión sea cerrada. Y, finalmente, tanto la cabeza como la cola de una lista no vacía serán variables.

Con esta transformación se consigue que toda subexpresión se trate de forma compartida, y por lo tanto sea evaluada una única vez. En consecuencia, nos hemos inclinado por la opción de sacar el máximo partido de las expresiones compartidas. De este modo estamos ante un ejemplo de evaluación basada en la *pereza completa* (definida en el glosario del Apéndice B).

Ejemplo 4.7 *Expresiones compartidas.*

Si una λ-abstracción ligada a una variable es demandada en más de una ocasión, la utilización de su cuerpo lleva a su evaluación cada vez que es aplicada. Sin embargo, si cada una de sus subexpresiones es sustituida por una variable a la que se liga, esta duplicación de trabajo no tiene lugar:

$$\begin{array}{l}\texttt{let } x = \backslash x_1.x_1\,((\backslash z.z)3),\\ \qquad w = (\backslash z.z)\#(x\,(\backslash t.t))\\ \texttt{in } x\,(\backslash z.z)\end{array}$$

En esta expresión, y supuesto que se disponen de recursos suficientes para evaluar el proceso inmerso, la expresión $(\backslash z.z)3$ sería evaluada dos veces. Sin embargo, si dicha

expresión estuviera ligada a una variable, dicha duplicación no tendría lugar:

$$\begin{array}{l}\texttt{let } x = \backslash x_1.x_1\, y,\\ \qquad y = (\backslash z.z)\, 3,\\ \qquad w = (\backslash z.z)\texttt{\#}(x\,(\backslash t.t))\\ \texttt{ in } x\,(\backslash z.z)\end{array}$$

□

Además, el que los argumentos de las aplicaciones sean variables simplifica las reglas y las funciones semánticas, al no tener que introducir variables nuevas para modelizar la pereza. La sintaxis restringida queda recogida en la Figura 4.4.

Jauja restringido		
E	$::=\quad x$	variable
	$\mid\quad \backslash x.E$	λ-abstracción
	$\mid\quad x_1\, x_2$	aplicación
	$\mid\quad x_1\texttt{\#}x_2$	creación de procesos
	$\mid\quad \texttt{let } \{x_i = E_i\}_{i=1}^{n} \texttt{ in } x$	declaración local
	$\mid\quad \texttt{new}(y, x)E$	canal dinámico
	$\mid\quad x\,!\,y_1 \texttt{ par } y_2$	conexión dinámica
	$\mid\quad x_1 \bowtie x_2$	mezcla
	$\mid\quad \Lambda[x_1 : x_2].E_1 [\![E_2$	Λ-abstracción
	$\mid\quad L$	lista
L	$::=\quad \texttt{nil}$	lista vacía
	$\mid\quad [x_1 : x_2]$	lista no vacía

Figura 4.4: Jauja: sintaxis restringida

El algoritmo de normalización de expresiones es sencillo, salvo por el especial tratamiento que requieren las expresiones de creación de procesos. La Figura 4.5 muestra dicho algoritmo, donde cada variable introducida es fresca.

En los casos básicos, i.e. la expresión a normalizar es una *variable* o la *lista vacía* `nil`, el proceso de normalización no realiza transformación alguna. En el resto de expresiones, el algoritmo de normalización recurre a la recursión. Si la expresión es una *λ-abstracción*, la expresión normalizada es la misma abstracción a excepción del cuerpo, que se normaliza.

Para la *aplicación funcional* se distinguen cuatro casos, con el fin de no introducir variables de normalización innecesarias.

La normalización de una *declaración local* ha de mantener en el nivel superior todas las creaciones de proceso que allí se hallen.

Ejemplo 4.8 *Normalización de una declaración local de variables.*

$$\texttt{let } x = (\backslash t.t)\texttt{\#}((\backslash s.s)4), y = 3 \texttt{ in } y$$

$$\mathsf{norm}\ x = x$$

$$\mathsf{norm}\ (\backslash x.E) = \backslash x.(\mathsf{norm}\ (E))$$

$$\mathsf{norm}\ (E_1\ E_2) = \begin{cases} E_1\ E_2, & \text{si } E_1, E_2 \in \mathit{Var}; \\ \mathtt{let}\ x_2 = (\mathsf{norm}\ E_2), z = E_1\ x_2\ \mathtt{in}\ z, & \text{si } E_1 \in \mathit{Var} \wedge E_2 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), z = x_1\ E_2\ \mathtt{in}\ z, & \text{si } E_2 \in \mathit{Var} \wedge E_1 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), x_2 = (\mathsf{norm}\ E_2), z = x_1\ x_2\ \mathtt{in}\ z, & \text{e.o.c.} \end{cases}$$

$$\mathsf{norm}\ (\mathtt{let}\ \{x_i = E_i\}_{i=1}^n\ \mathtt{in}\ E) = \mathtt{let}\ \mathit{lvs}\ \mathtt{in}\ x$$
$$\quad \mathtt{donde}\ \mathit{lvs} = \mathsf{normLVars}\ (\{x_i = E_i\}_{i=1}^n \cup \{x = E\})$$

$$\mathsf{norm}\ (E_1 \# E_2) = \mathsf{normCrea}\ (E_1 \# E_2)$$

$$\mathsf{norm}\ (\mathtt{new}(y, x)E) = \mathtt{new}(y, x)(\mathsf{norm}\ (E))$$

$$\mathsf{norm}\ (x\,!\,E_1\ \mathtt{par}\ E_2) = \begin{cases} x\,!\,E_1\ \mathtt{par}\ E_2, & \text{si } E_1, E_2 \in \mathit{Var}; \\ \mathtt{let}\ x_2 = (\mathsf{norm}\ E_2), z = x\,!\,E_1\ \mathtt{par}\ x_2\ \mathtt{in}\ z, & \text{si } E_1 \in \mathit{Var} \wedge E_2 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), z = x\,!\,x_1\ \mathtt{par}\ E_2\ \mathtt{in}\ z, & \text{si } E_2 \in \mathit{Var} \wedge E_1 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), x_2 = (\mathsf{norm}\ E_2), z = x\,!\,x_1\ \mathtt{par}\ x_2\ \mathtt{in}\ z, & \text{e.o.c.} \end{cases}$$

$$\mathsf{norm}\ (E_1 \bowtie E_2) = \begin{cases} E_1 \bowtie E_2, & \text{si } E_1, E_2 \in \mathit{Var}; \\ \mathtt{let}\ x_2 = (\mathsf{norm}\ E_2), z = E_1 \bowtie x_2\ \mathtt{in}\ z, & \text{si } E_1 \in \mathit{Var} \wedge E_2 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), z = x_1 \bowtie E_2\ \mathtt{in}\ z, & \text{si } E_2 \in \mathit{Var} \wedge E_1 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), x_2 = (\mathsf{norm}\ E_2), z = x_1 \bowtie x_2\ \mathtt{in}\ z, & \text{e.o.c.} \end{cases}$$

$$\mathsf{norm}\ (\Lambda[x_1 : x_2].E_1 [\![E_2) = \Lambda[x_1 : x_2].(\mathsf{norm}\ (E_1))[\![(\mathsf{norm}\ (E_2))$$

$$\mathsf{norm}\ (\mathtt{nil}) = \mathtt{nil}$$

$$\mathsf{norm}\ ([E_1 : E_2]) = \begin{cases} [E_1 : E_2], & \text{si } E_1, E_2 \in \mathit{Var}; \\ \mathtt{let}\ x_2 = (\mathsf{norm}\ E_2), z = [E_1 : x_2]\ \mathtt{in}\ z, & \text{si } E_1 \in \mathit{Var} \wedge E_2 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), z = [x_1 : E_2]\ \mathtt{in}\ z, & \text{si } E_2 \in \mathit{Var} \wedge E_1 \notin \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ E_1), x_2 = (\mathsf{norm}\ E_2), z = [x_1 : x_2]\ \mathtt{in}\ z, & \text{e.o.c.} \end{cases}$$

Figura 4.5: Jauja: normalización de expresiones

La creación de proceso $(\backslash t.t)\#((\backslash s.s)\ 4)$ debe permanecer en el nivel superior. Es por ello que una normalización trivial en la que cada variable quedaría ligada a la normalización de la expresión de partida no bastaría:

$$\begin{array}{l} \mathtt{let}\ x = \mathtt{let}\ x_1 = \mathtt{let}\ x_3 = 4, \\ \qquad\qquad\qquad\qquad\ x_4 = \backslash s.s, \\ \qquad\qquad\qquad\qquad\ x_5 = x_4 x_3 \\ \qquad\qquad\qquad\ \mathtt{in}\ x_5, \\ \qquad\qquad\ x_0 = \backslash t.t, \\ \qquad\qquad\ x_6 = x_0 \# x_1, \\ \qquad\ \mathtt{in}\ x_6 \\ \quad y = 3 \\ \mathtt{in}\ y \end{array}$$

pues se perderá la creación en el nivel superior. La solución pasa por unificar los dos `let` más externos:

$$\begin{array}{l}\texttt{let } x = x_6,\\ \quad x_1 = \texttt{let } x_3 = 4,\\ \qquad\qquad x_4 = \backslash s.s,\\ \qquad\qquad x_5 = x_4 x_3\\ \qquad \texttt{in } x_5,\\ \quad x_0 = \backslash t.t,\\ \quad x_6 = x_0 \texttt{\#} x_1,\\ \quad y = 3\\ \texttt{in } y\end{array}$$

De este modo la creación de proceso permanece en el nivel superior.

□

Así, la normalización de una declaración local de variables actúa sobre un conjunto de ligaduras, incluyendo la de una nueva variable al cuerpo de la declaración. La función auxiliar normLVars normaliza las ligaduras de variables del `let`, para lo cual toma como argumento dicho conjunto de ligaduras, actuando de modo diferente según la forma de la expresión local:

- Creación de proceso: tiene que unificar los `let` como en el ejemplo anterior, para lo que usamos la función unifLet.
- Otro tipo de expresión: la variable se liga a la normalización de la expresión local.

Con estas restricciones, la normalización de un conjunto de ligaduras se formaliza como sigue:

$$\text{normLVars } lvs = \begin{cases} \emptyset & \text{si } lvs = \emptyset \\ (\text{unifLet } (\text{normLVar } (x = E))) \cup (\text{normLVars } lvs'), & \text{si } lvs = \{x = E\} \cup lvs' \ E \equiv E_1 \texttt{\#} E_2; \\ \text{normLVar } (x = E) \cup (\text{normLVars } lvs'), & \text{si } lvs = \{x = E\} \cup lvs' \ E \not\equiv E_1 \texttt{\#} E_2. \end{cases}$$

$$\text{unifLet } (x = E) = \begin{cases} \{x = y\} \cup lvs, & \text{si } E \equiv \texttt{let } lvs \texttt{ in } y; \\ \{x = E\}, & \text{e.o.c.} \end{cases}$$

$$\text{normLVar } (x = E) = (x = (\text{norm } E))$$

La normalización de una creación de proceso tampoco es sencilla, ya que es necesario mantener la posibilidad de creación encadenada:

Ejemplo 4.9 *Normalización de creaciones de proceso.*

$$(\backslash x.x)\texttt{\#}((\backslash y.y)\texttt{\#}(3))$$

En esta expresión, la creación de proceso más externa habilita la más interna. Por ello, la simple normalización de ambas partes de la creación y su combinación mediante # no

es suficiente:

$$\begin{array}{l}\texttt{let } x_0 = \backslash x.x,\\ \quad x_3 = \texttt{let } x_1 = \backslash y.y,\\ \qquad\qquad\quad x_2 = 3,\\ \qquad\qquad\quad x_5 = x_1 \texttt{\#} x_2\\ \qquad\quad \texttt{in } x_5,\\ \quad x_4 = x_0 \texttt{\#} x_3\\ \texttt{in } x_4\end{array}$$

Como se observa, la creación de proceso más interna no aparecería en el nivel superior. Sin embargo, elevando todas las creaciones al nivel superior se obtiene:

$$\begin{array}{l}\texttt{let } x_0 = \backslash x.x,\\ \quad x_1 = \backslash y.y,\\ \quad x_2 = 3,\\ \quad x_3 = x_1 \texttt{\#} x_2,\\ \quad x_4 = x_0 \texttt{\#} x_3\\ \texttt{in } x_4\end{array}$$

que corresponde al comportamiento deseado.

□

A la vista del ejemplo, la normalización de la creación de procesos queda formalizada como sigue:

$$\mathsf{normCrea}\ (E_1\texttt{\#}E_2) = \begin{cases} E_1\texttt{\#}E_2, & \text{si } E_1, E_2 \in \mathit{Var};\\ \texttt{let } (\{y = E_1\texttt{\#}x\} \cup \mathit{lvs}) \texttt{ in } y, & \text{si } E_1 \in \mathit{Var} \wedge E_2 \equiv (E_2^1\texttt{\#}E_2^2);\\ \quad \texttt{donde let } \mathit{lvs} \texttt{ in } x = \mathsf{norm}\ (E_2^1\texttt{\#}E_2^2) & \\ \texttt{let } \{y = (\mathsf{norm}\ E_2), z = E_1\texttt{\#}y\} \texttt{ in } z, & \text{si } E_1 \in \mathit{Var} \wedge E_2 \not\equiv (E_2^1\texttt{\#}E_2^2);\\ \texttt{let } (\{y = \mathit{creaPpal}\} \cup \mathit{bs}) \texttt{ in } y, & \text{e.o.c.}\\ \quad \texttt{donde } (\mathit{vars1}, \mathit{exprs1}) = \mathsf{dcCrea}\ (E_1\texttt{\#}E_2) & \\ \qquad\quad (\mathit{vars2}, \mathit{creas1}) = \mathsf{compCrea}\ \mathit{vars1}\ \mathit{vars1}\ [\,] & \\ \qquad\quad \mathit{creaPpal} = \texttt{head } \mathit{creas1} & \\ \qquad\quad \mathit{creas2} = \texttt{tail } \mathit{creas1} & \\ \qquad\quad \mathit{exprs2} = \mathsf{map\ norm}\ \mathit{exprs1} & \\ \qquad\quad \mathit{exprCrea} = \mathit{creas2} +\!\!+ \mathit{exprs2} & \\ \qquad\quad \mathit{bs} = \mathsf{creLetB}\ \mathit{vars2}\ \mathit{exprCrea} & \end{cases}$$

Los pasos simples de la normalización son:

1. (dcCrea)

$$\mathsf{dcCrea}\ E_1\texttt{\#}E_2 = \begin{cases} (\mathit{vars} +\!\!+ [x], \mathit{exprs} +\!\!+ [E_1]), & \text{si } E_2 \equiv E_2^1\texttt{\#}E_2^2;\\ \quad \texttt{donde } (\mathit{vars}, \mathit{exprs}) = \mathsf{dcCrea}\ (E_2^1\texttt{\#}E_2^2) & \\ ([y, x], [E_2, E_1]), & \text{e.o.c.}\end{cases}$$

Descomposición de la cadena de creaciones de proceso en una lista de variables y otra de subexpresiones asociadas a dichas variables.

2. (compCrea): recomposición de las creaciones, de modo que aparezcan las creaciones en forma normalizada:

$$\mathsf{compCrea}\ vars1\ vars2\ creas = \begin{cases} (vars2,\ creas), & \text{si } vars1 = [\,]; \\ (vars2, [y\#x] +\!\!+ creas), & \text{si } vars1 = [x, y]; \\ \mathsf{compCrea}\ (z : xs)\,(z : vars2)\,((y\#x) : creas), & \text{si } vars1 = (x : (y : xs)). \end{cases}$$

   Se manejan dos listas de variables, que inicialmente se corresponden con la obtenida en el paso anterior; la primera se va "destruyendo" para ir componiendo #-expresiones entre variables; y en la segunda lista, junto a las variables que tuviéramos en un principio, se van añadiendo las nuevas, que son necesarias para poder generar todas las expresiones de creación. Esta segunda lista se devuelve al final conteniendo todas las variables de la declaración local resultado de la segunda normalización. En el Ejemplo 4.9 la lista inicial es $[x_2, x_1, x_0]$ y la final $[x_3, x_2, x_1, x_0]$. En tanto que la lista de creaciones es $[x_0\#x_3, x_1\#x_2]$.

3. La cabeza de la lista de las creaciones recompuestas es la creación principal.

4. La cola contiene el resto de la cadena de creaciones normalizadas.

5. Normalización de las subexpresiones que componían las creaciones iniciales.

6. Para crear la declaración local final, cada creación y cada subexpresión han de ser ligadas a la variable correspondiente.

7. (creLetB): se crea el `let` con las ligaduras adecuadas.

$$\mathsf{creLetB}\ vars\ exprs = \begin{cases} [\,], & \text{si } vars = [\,] \text{ y } exprs = [\,]; \\ \{x = e\} \cup (\mathsf{creLetB}\ xs\ es), & \text{si } vars = (x : xs) \text{ y } exprs = (e : es). \end{cases}$$

Este proceso de normalización de creaciones encadenadas —del tipo de la que aparece en el Ejemplo 4.9— siempre termina, pues cualquier expresión de Jauja está formada por un número finito de símbolos, de modo que el número de #-expresiones encadenadas es finito también.

El resto de las normalizaciones siguen el patrón de la aplicación funcional, pero con la estructura adecuada, y ya han sido incluidas en la Figura 4.5.

Llegados a este punto tenemos a Jauja listo para definir la semántica de sus expresiones. En las Partes II y III del trabajo se presentan las semánticas operacional y denotacional, respectivamente.

PARTE II

SEMÁNTICA OPERACIONAL

CAPÍTULO 5

Fundamentos de nuestra semántica

Estudia el pasado si quieres pronosticar el futuro.
Confucio

De la introducción a Jauja realizada en el Capítulo 4 se deduce que nuestra primera, que no última, intención es construir una semántica operacional que modelice las características fundamentales allí descritas, a saber, *call-by-need*, pereza, y paralelismo en dos niveles: dentro de un proceso y entre procesos.

Los dos trabajos que han inspirado la semántica operacional que detallaremos en el Capítulo 6, son la semántica natural de Launchbury, presentada en [Lau93], y la semántica operacional para evaluar en un ámbito de pereza y paralelismo, desarrollada por Baker-Finch y otros en [BKT00].

5.1. Semántica natural de Launchbury

Según su autor, se trata de "una semántica operacional para evaluación paralela en la que se incluye de manera precisa un modelo para compartir ligaduras". Su estructura computacional básica es un conjunto de ligaduras que denominaremos *heap*. Puede decirse que su nivel de abstracción se encuentra en un punto intermedio entre la semántica denotacional de [Abr90] y una semántica operacional basada en el funcionamiento de una máquina abstracta.

La definición de la semántica consta de dos fases. La primera, en el estilo de la descrita en la Sección 4.3, es una transformación estática de una λ-expresión en otra normalizada en la que se consigue que las subexpresiones sean compartidas. La segunda es una semántica dinámica que dota de significado a estas expresiones normalizadas.

El lenguaje al que Launchbury dota de semántica queda especificado en la Figura 5.1.

Lenguaje de [Lau93]		
E ::=	x	variable
|	$\backslash x.E$	λ-abstracción
|	$E_1 E_2$	aplicación
|	`let` $\{x_i = E_i\}_{i=1}^{n}$ `in` E	declaracion local

Figura 5.1: Launchbury: sintaxis

Éste es un λ-cálculo extendido con una declaración local de variables potencialmente recursiva. La incorporación de la construcción `let` en este cálculo no es mero azúcar sintáctico, pues permite que existan definiciones cíclicas de variables, como sucede en la mayoría de las implementaciones de los lenguajes funcionales.

5.1.1. Normalización

Debido a la ausencia de paralelismo, el proceso de transformación sintáctica de expresiones es mucho más sencillo que el detallado para Jauja. Sin embargo, la normalización consta de dos partes, de las cuales la primera no ha sido necesaria para Jauja.

Dada una expresión, E, el *primer paso* de normalización la convierte en otra expresión, $\hat{E}$, resultado de renombrar todas las variables ligadas de E empleando nombres de variable nuevos, de modo que todas las variables ligadas sean distintas; además, como consecuencia del renombramiento, en $\hat{E}$ no importa el ámbito de las variables, pudiéndose incorporar así todas las variables en el mismo *heap* de almacenamiento al proceder a la evaluación del programa. En Jauja esta fase no fue necesaria porque, como veremos más adelante, nosotros realizamos el renombramiento cuando incorporamos variables locales en el *heap* —en el caso de la semántica operacional —o en el entorno —en el caso de la semántica denotacional—.

El *segundo paso* de la normalización tiene por objeto que todas las aplicaciones sean de una expresión a una variable. El motivo de esta restricción es simplificar la regla dinámica de aplicación para introducir pereza y compartir ligaduras. La sintaxis restringida se detalla en la Figura 5.2.

El proceso de normalización comienza entonces por transformar la expresión inicial E en $\hat{E}$, aplicando simplemente la habitual α-conversión utilizando variables frescas para renombrar. La segunda fase consiste en la aplicación del algoritmo de la Figura 5.3.

Este simple algoritmo de normalización consiste en sustituir por variables los argu-

Lenguaje Restringido de [Lau93]		
E	$::= \quad x$	variable
	$\mid \quad \backslash x.E$	λ-abstracción
	$\mid \quad E\ x$	aplicación
	$\mid \quad \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	declaracion local

Figura 5.2: Launchbury: sintaxis restringida

$$\mathsf{norm}\ x = x$$

$$\mathsf{norm}\ (\backslash x.E) = \backslash x.(\mathsf{norm}\ (E))$$

$$\mathsf{norm}\ (E_1\ E_2) = \begin{cases} (\mathsf{norm}\ (E_1))\ E_2, & \text{si } E_2 \in \mathit{Var}; \\ \texttt{let}\ x_2 = (\mathsf{norm}\ (E_2))\ \texttt{in}\ (\mathsf{norm}\ (E_1))\ x_2, & \text{e.o.c.} \end{cases}$$

$$\mathsf{norm}\ (\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E) = \texttt{let}\ \{x_i = (\mathsf{norm}\ (E_i))\}_{i=1}^{n}\ \texttt{in}\ (\mathsf{norm}\ (E))$$

Figura 5.3: Launchbury: normalización de expresiones

mentos de las aplicaciones y expresar esta sustitución utilizando expresiones `let`.

5.1.2. Semántica natural

Los elementos sobre los que se define la semántica son *heaps*, valores y juicios. Los primeros, representados por Γ, Δ y Θ, son ligaduras de variables, todas distintas, a expresiones, o dicho de otro modo, funciones parciales de variables a expresiones. Los valores, incluidos en el conjunto *Val* y representados por W, son las λ-abstracciones, es decir, valores *whnf*. Finalmente, los juicios tienen la forma:

$$\Gamma : E \Downarrow \Delta : W$$

cuya lectura es: "el término E, en el contexto del conjunto de ligaduras Γ, se reduce al valor W, junto con el posiblemente modificado conjunto de ligaduras Δ". Las modificaciones surgen de la posibilidad de incorporar nuevas ligaduras en el *heap* y de la actualización con los valores obtenidos de su evaluación de ligaduras aún sin evaluar.

Una demostración de un juicio se corresponde con una secuencia de reducción. Una de estas demostraciones puede no lograrse por dos motivos:

- No existe una demostración finita que pruebe que la reducción es válida.
- No existe una regla aplicable a una subparte de la demostración (hecho denominado como agujero negro).

Finalmente, para definir la semántica se construyen las reglas de reducción de la Figura 5.4.

$$\Gamma : \backslash x.E \Downarrow \Gamma : \backslash x.E \qquad \textbf{(lambda)}$$

$$\frac{\Gamma : E \Downarrow \Delta : W}{\Gamma + x \mapsto E : x \Downarrow \Delta + x \mapsto W : \hat{W}} \qquad \textbf{(variable)}$$

$$\frac{\Gamma : E \Downarrow \Delta : \backslash y.E' \qquad \Delta : E'[x/y] \Downarrow \Theta : W}{\Gamma : E\, x \Downarrow \Theta : W} \qquad \textbf{(application)}$$

$$\frac{\Gamma + \{x_i \mapsto E_i\}_{i=1}^{n} : E \Downarrow \Delta : W}{\Gamma : \texttt{let } \{x_i = E_i\}_{i=1}^{n} \texttt{ in } E \Downarrow \Delta : W} \qquad \textbf{(let)}$$

Figura 5.4: Launchbury: reglas de reducción

La primera regla, LAMBDA, es en realidad un axioma, que permitirá finalizar las demostraciones. Simplemente expresa que un valor en *whnf* es la meta que persigue la evaluación de toda expresión, por lo que una λ-expresión se reduce a sí misma, sin modificar el *heap*.

La regla VARIABLE es la que la realiza la reducción de variables: si la expresión ligada a la variable x se reduce a un valor *whnf*, entonces la variable se reduce a dicho valor, y el resultado de la reducción es el valor renombrado adecuadamente, para facilitar su posterior uso. El *heap* puede ser modificado en esta reducción, conservándose dicha modificación en el *heap* de la conclusión. Es en este punto donde queda recogido cómo se comparte en el modelo, pues para poder evaluar una variable se ha requerido que exista en el *heap* una ligadura con dicha variable como lado izquierdo, y tras la reducción la variable ha quedado ligada al valor, con lo que éste puede ser utilizado para evaluar otra ligadura.

La regla APPLICATION establece que dada una aplicación funcional normalizada, $E\,x$, si la primera parte se reduce a una λ-abstracción, y el cuerpo de esta abstracción, donde el parámetro formal y se sustituye por el actual x, se reduce a un valor *whnf*, entonces la aplicación completa se reduce a dicho valor. Vemos que éste es el momento en el que la normalización se convierte en una estrategia útil, pues no se produce una sustitución de una variable por un valor, sino que se sustituye una variable por otra variable ligada a un valor.

Para la declaración local de variables, la regla LET establece que si el cuerpo se puede reducir en un *heap* donde se han incluido las ligaduras de las variables locales, dando lugar a un valor *whnf*, entonces la expresión completa se reduce al mismo valor en el *heap* sin estas ligaduras nuevas. La normalización previa posibilita que las variables nuevas no tengan que ser renombradas para incorporarlas en el *heap*. Pero no es sólo la normalización previa quien garantiza la validez de esta acción, pues el renombramiento que se realiza en la regla VARIABLE es imprescindible para permitir la inclusión de ligaduras de este tipo.

A partir de esta semántica, Baker-Finch y otros definen en [BKT00] su semántica

operacional para expresar de manera conjunta pereza y paralelismo.

5.2. Semántica operacional de GpH

En esta sección se estudiará la semántica operacional para el lenguaje Glasgow Parallel Haskell (GpH) presentada en [BKT00], y que es una revisión de [HBTK99]. A partir de la semántica que a continuación discutiremos, fue desarrollada posteriormente otra en [Bak00] cuyo nivel de abstracción es intermedio entre el de una máquina abstracta y el conseguido aquí. Realicemos primero una exposición general de las características de GpH.

5.2.1. Una breve panorámica sobre GpH

GpH es un lenguaje paralelo cuyo núcleo funcional está formado por Haskell. GpH es evaluado siguiendo las normas *call-by-need* o evaluación perezosa, es decir, se evalúa a *whnf* tomando como *redex* aquél más externo y que esté más a la izquierda.

GpH extiende Haskell introduciendo paralelismo mediante anotaciones que indican las zonas de código que pueden ser ejecutadas en paralelo, siempre que se disponga de recursos suficientes para hacerlo. Se utiliza, pues, el enfoque de paralelismo semi-explícito visto en la Sección 2.2.2.3.

Las dos anotaciones introducidas son `par` y `seq`. La primera de ellas indica la posibilidad de ejecución en paralelo, mientras que la segunda fuerza una evaluación secuencial. El significado, o resultado de evaluar, de ambas construcciones queda reflejado en las siguientes "ecuaciones":

$$E_1 \text{ 'par' } E_2 = E_2$$
$$\bot \text{ 'seq' } E_2 = \bot$$
$$E_1 \text{ 'seq' } E_2 = E_2$$

Así, la expresión E_1 'par' E_2 indica que se evalúen E_1 y E_2 simultáneamente en procesadores distintos, siempre que los recursos del sistema lo permitan. Por su parte, en la expresión E_1 'seq' E_2 se realiza primero la evaluación de E_1 a *whnf*, pasando después a la evaluación de E_2; este orden de evaluación rompe la pereza del lenguaje y fuerza la evaluación de expresiones a *whnf*. Como se exige la evaluación a *whnf* del primer argumento de `seq`, si éste está indefinido, la expresión completa también lo estará; esto no ocurre con `par`, pues la evaluación de los dos argumentos se realiza, en principio, de manera independiente. En ambos casos, el valor resultante de la evaluación de la expresión es el calculado por el segundo argumento, E_2. En GpH se establece una clara separación entre el algoritmo definido por una función y la especificación de su comportamiento dinámico empleando estrategias de evaluación definidas a partir de estos combinadores `par` y `seq` [THLP98].

A diferencia de Eden (y Jauja), GpH no dispone de un concepto explícito de proceso

ni de canales para realizar comunicaciones entre las distintas partes que se evalúan en paralelo.

El núcleo sintáctico de GpH que vamos a usar es, básicamente, el mismo que fue empleado en su momento en [BKT00]. Su sintaxis queda recogida en la Figura 5.5.

E	$::=$	x	variable
	$\mid$	$\backslash x.E$	λ-abstracción
	$\mid$	$E_1\ E_2$	aplicación
	$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	declaración local
	$\mid$	$E_1\ \texttt{'seq'}\ E_2$	composición secuencial
	$\mid$	$E_1\ \texttt{'par'}\ E_2$	composición paralela

Figura 5.5: GpH-core: sintaxis

Las construcciones sintácticas básicas son las mismas que contiene Jauja, no en vano el núcleo funcional de ambos lenguajes es el mismo. Por tanto, la evaluación de variables, abstracciones, aplicaciones funcionales y declaración local de variables será idéntica. La nota de color la ponen las composiciones secuencial y paralela.

5.2.2. Normalización

Al igual que en [Lau93], en [BKT00] se realiza un proceso de normalización previo, para evitar choques en los nombres de las variables y para simplificar las reglas de reducción, con las mismas dos fases: inicial de renombramiento y siguiente de transformación de expresiones a una sintaxis restringida. Dicha sintaxis incluye sustituciones de expresiones por variables, en la misma línea que siguió Launchbury. Esta sintaxis restringida de GpH-core queda recogida en la Figura 5.6.

E	$::=$	x	variable
	$\mid$	$\backslash x.E$	λ-abstracción
	$\mid$	$E\ x$	applicación
	$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	declaración local
	$\mid$	$E_1\ \texttt{'seq'}\ E_2$	composición secuencial
	$\mid$	$x\ \texttt{'par'}\ E$	composición paralela

Figura 5.6: GpH-core: sintaxis restringida

Al igual que en la semántica natural de Launchbury, la subexpresión argumento de la aplicación funcional es ahora una variable. En cuanto a las construcciones propias de GpH, solamente se introducen modificaciones en la composición paralela, cambiando la primera expresión por una variable, para no tener que introducirla en la regla semántica correspondiente. El algoritmo de normalización, obtenido a partir del presentado en [HBTK99], se detalla en la Figura 5.7.

Difieren este algoritmo y el de la semántica de Launchbury en las construcciones propias de GpH, de las cuales la composición secuencial no sustituye variables y simple-

$$\mathsf{norm}\ x = x$$

$$\mathsf{norm}\ (\backslash x.E) = \backslash x.(\mathsf{norm}\ (E))$$

$$\mathsf{norm}\ (E_1\ E_2) = \begin{cases} (\mathsf{norm}\ (E_1))\ E_2, & \text{si } E_2 \in \mathit{Var}; \\ \mathtt{let}\ x_2 = (\mathsf{norm}\ (E_2))\ \mathtt{in}\ (\mathsf{norm}\ (E_1))\ x_2, & \text{e.o.c.} \end{cases}$$

$$\mathsf{norm}\ (\mathtt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \mathtt{in}\ E) = \mathtt{let}\ \{x_i = (\mathsf{norm}\ (E_i))\}_{i=1}^{n}\ \mathtt{in}\ (\mathsf{norm}\ (E))$$

$$\mathsf{norm}\ (E_1\ \text{'}\mathtt{par}\text{'}\ E_2) = \begin{cases} E_1\ \text{'}\mathtt{par}\text{'}\ (\mathsf{norm}\ (E_2)), & \text{si } E_1 \in \mathit{Var}; \\ \mathtt{let}\ x_1 = (\mathsf{norm}\ (E_1))\ \mathtt{in}\ x_1\ \text{'}\mathtt{par}\text{'}\ (\mathsf{norm}\ (E_2)), & \text{e.o.c.} \end{cases}$$

$$\mathsf{norm}\ (E_1\ \text{'}\mathtt{seq}\text{'}\ E_2) = (\mathsf{norm}\ (E_1))\ \text{'}\mathtt{seq}\text{'}\ (\mathsf{norm}\ (E_2))$$

Figura 5.7: GPH-CORE: normalización de expresiones

mente normaliza las subexpresiones, y en la composición paralela, donde se introduce una declaración local que liga una nueva variable a la expresión del lado izquierdo normalizada, dejando sin ligar la expresión del lado derecho, pero aplicándole la normalización.

5.2.3. Semántica operacional

La semántica de [BKT00] presenta una diferencia fundamental con respecto a la de Launchbury: es una semántica de más bajo nivel o más cercana al funcionamiento de una máquina abstracta en la que se evaluarían los programas, hecho que queda patente al utilizar no ya *heaps* para recoger las ligaduras de ejecución, sino el etiquetado de éstas con los estados de ejecución por los que pasan. No es, por tanto, una semántica que se limite a indicar el proceso de reducción de un programa, aunque es una extensión de la semántica natural expuesta en la Sección 5.1 para incluir las dos primitivas que caracterizan GPH: `par` y `seq`. En una versión anterior del trabajo, [BKHT99], se demuestra que esta semántica para GPH es correcta con respecto a la de [Lau93] si se tiene un único procesador; además, ambas se corresponden, si a la no paralela se le añaden las dos primitivas anteriores.

Recordemos que el carácter paralelo del lenguaje se refleja en el hecho de contar con varias ligaduras de ejecución (asimilables a los procesos de Jauja) que pueden ser ejecutables al mismo tiempo, de modo que, si el sistema dispone de procesadores libres, pueden estar ejecutándose simultáneamente, con lo que todas ellas pasarían a ser ligaduras *activas*.

Los elementos básicos de esta semántica van a ser también ligaduras y *heaps*. Sin embargo, en este caso las ligaduras van a estar etiquetadas de la forma siguiente:

Inactiva: o bien la variable está ligada a un valor *whnf* o bien esta variable no ha sido demandada aún.

Bloqueada: la variable ha sido demandada y ha comenzado a ser evaluada la expresión asociada, pero se ha llegado a un punto en el que necesita el valor que se asociará a otra variable que aún no se encuentra disponible.

Ejecutable: corresponde a una ligadura cuyo valor ha sido demandado, que se está intentando obtener, pero que no hay recursos suficientes en el sistema para ejecutar la evaluación.

Activa: la variable ha sido demandada, existen recursos suficientes en el sistema y, en consecuencia, se está procediendo a su evaluación.

Todas las ligaduras se reúnen en un *heap*, H; el hecho de estar todas juntas hace que en las políticas de planificación de los procesadores se consideren todas las clausuras en conjunto y que se compartan expresiones entre todas ellas.

Así, las ligaduras son de la forma:

$x \overset{\alpha}{\mapsto} E$, donde $\alpha ::= I$ (Inactiva) $\mid B$ (Bloqueada) $\mid R$ (Ejecutable) $\mid A$ (Activa)

siendo E un término normalizado y α el estado de la ligadura.

Dado un *heap* H, se denota con H^α al subconjunto de H formado por todas las ligaduras etiquetadas con α.

La semántica operacional de GpH-core viene determinada por transiciones entre *heaps*. Estas transiciones son de dos tipos: unas involucran a una sola ligadura, es decir, no consideran paralelismo (paso simple, en la Sección 5.2.3.1), las otras tratan el paralelismo realizando transiciones considerando a todas las ligaduras que pueden ejecutarse a la vez (paso múltiple, en la Sección 5.2.3.2).

Con estos dos tipos de reglas se establece una secuencia de reducción:

$$(main \overset{A}{\mapsto} E) \implies \ldots \implies (H, main \overset{I}{\mapsto} W)$$

donde se emplea *main* como el identificador del programa, y siendo W una expresión de GpH-core en *whnf*.

5.2.3.1. Transiciones de paso simple

Las transiciones de paso simple describen cómo una ligadura activa va a ser reducida en el contexto de un *heap*. Las reglas que describen estas transiciones se muestran en la Figura 5.8. En todas las reglas se emplea la siguiente notación: $(H, x \overset{\alpha}{\mapsto} E)$ es un *heap* en el que se separa la ligadura de x del resto, y $H : x \overset{A}{\mapsto} E$ indica que es sobre la ligadura (activa) de x sobre la que se aplicará la regla de paso simple en cuestión. Las transiciones pueden estar multietiquetadas, por ejemplo, $x \overset{ABR}{\mapsto} E$ indica que el estado puede ser A, B o R, pero en ningún caso I. El *heap* de la derecha de una transición incluye sólo aquellas ligaduras que varían o que son nuevas, lo cual no quiere decir que el resto desaparezca, simplemente que permanece invariable.

La regla variable es la única en la que es necesario realizar el renombramiento de variables, $\hat{W}$, pues si en el seno del valor se encuentra una declaración local de variables,

$$(H, x \overset{I}{\mapsto} W) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{A}{\mapsto} \hat{W}) \qquad \textbf{(variable)}$$

$$(H, x \overset{I}{\mapsto} E) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{B}{\mapsto} x, x \overset{R}{\mapsto} E) \qquad \textbf{(blocking-I)}$$

$$(H, x \overset{ABR}{\mapsto} E) : z \overset{A}{\mapsto} x \longrightarrow (z \overset{B}{\mapsto} x) \qquad \textbf{(blocking)}$$

$$H : z \overset{A}{\mapsto} z \longrightarrow (z \overset{B}{\mapsto} z) \qquad \textbf{(blackhole)}$$

$$H : z \overset{A}{\mapsto} (\backslash y.E)x \longrightarrow (z \overset{A}{\mapsto} E[x/y]) \qquad \textbf{(}\beta\textbf{-reduction)}$$

$$\frac{H : z \overset{A}{\mapsto} E \longrightarrow (H', z \overset{\alpha}{\mapsto} E')}{H : z \overset{A}{\mapsto} E\, x \longrightarrow (H', z \overset{\alpha}{\mapsto} E'\, x)} \qquad \textbf{(application)}$$

$$H : z \overset{A}{\mapsto} \texttt{let}\ x_1 = E_1, \ldots, x_n = E_n\ \texttt{in}\ E \longrightarrow (x_1 \overset{I}{\mapsto} E_1, \ldots, x_n \overset{I}{\mapsto} E_n, z \overset{A}{\mapsto} E) \qquad \textbf{(let)}$$

$$H : z \overset{A}{\mapsto} W\ \texttt{`seq`}\ E \longrightarrow (z \overset{A}{\mapsto} E) \qquad \textbf{(elim-seq)}$$

$$\frac{H : z \overset{A}{\mapsto} E_1 \longrightarrow (H', z \overset{\alpha}{\mapsto} E_1')}{H : z \overset{A}{\mapsto} E_1\ \texttt{`seq`}\ E_2 \longrightarrow (z \overset{\alpha}{\mapsto} E_1'\ \texttt{`seq`}\ E_2)} \qquad \textbf{(seq)}$$

$$(H, x \overset{ABR}{\mapsto} E_1) : z \overset{A}{\mapsto} x\ \texttt{`par`}\ E_2 \longrightarrow (z \overset{A}{\mapsto} E_2) \qquad \textbf{(elim-par)}$$

$$(H, x \overset{I}{\mapsto} E_1) : z \overset{A}{\mapsto} x\ \texttt{`par`}\ E_2 \longrightarrow (z \overset{A}{\mapsto} E_2, x \overset{R}{\mapsto} E_1) \qquad \textbf{(par)}$$

Figura 5.8: GpH-core: reglas de transición de paso simple

cuando este valor sea empleado en más de una ocasión —en la que dichas variables tengan que ser incorporadas al *heap*— se producirá un choque de nombres. Por lo demás, simplemente se copia el valor ligado a la variable x en la ligadura de la variable z.

Las tres reglas siguientes, blocking-I, blocking y blackhole, son la base para la introducción de ligaduras bloqueadas por demanda. En todas ellas se tiene una ligadura que no puede ser evaluada, bien porque necesita la evaluación de otra, bien porque es una autorreferencia. En la primera regla, una ligadura inactiva pasa a estado ejecutable por ser necesaria para evaluar la ligadura activa, i.e. la ligadura es demandada, y como la ligadura activa no se puede evaluar, pasa al estado de bloqueada. La diferencia entre la primera regla y la segunda de este grupo es el estado de la ligadura necesaria para evaluar la activa: en el segundo caso es no inactiva, o ya demandada anteriormente, por lo que solamente cambia el estado de la ligadura activa a bloqueada. La última de estas reglas incluye una ligadura en la que la variable se liga a ella misma, pasando esta ligadura a estar bloqueada, pues no es posible su evaluación más allá de lo ya conseguido.

Las siguientes reglas son las esenciales de la programación funcional: β-reduction y application. En la aplicación se observa el uso de la sintaxis restringida de GpH-core (Figura 5.6), ya que la aplicación se realiza sobre una variable. La pereza de la aplicación funcional queda reflejada en el hecho de que es la parte que dará lugar a la λ-abstracción la que se evalúa, postergando la del argumento hasta que éste sea estrictamente necesario. Además, este argumento sólo se evaluará una vez, ya que cuando sea necesario su uso en dos momentos distintos, en el primero se evaluará, pero en el segundo ya lo encontraremos evaluado obteniéndose por aplicación de la regla value

(recordemos que ya sucedía así en la semántica presentada en la Sección 5.1).

La regla LET introduce en el *heap* tantas ligaduras como variables locales se declaran, siendo el estado de todas estas nuevas ligaduras inactivo (pereza). Se activarán por separado, según vayan siendo reclamadas por alguna ligadura activa en un determinado momento. Ésta es la única regla que provoca un aumento del número de ligaduras en el *heap*. Al incorporarse todas las ligaduras correspondientes a las variables locales, logramos que el `let` sea recursivo, pues la evaluación de una de ellas podrá reclamar la de aquellas de las que dependa.

La semántica de la composición secuencial se define mediante dos reglas. En la primera, ELIM-SEQ, como el primer argumento del operador `seq` es ya un valor en *whnf*, sólo queda por evaluar el segundo argumento, que además es el que da lugar al valor ligado a la variable; estas condiciones son las que hacen posible la eliminación de `seq`. Cuando el primer argumento aún no está en *whnf* y puede ser evaluado con el resto del *heap*, se aplica SEQ procediéndose a dar un paso de reducción de toda la composición secuencial.

Finalmente, las últimas dos reglas determinan la semántica de la composición paralela. Se distinguen dos casos dependiendo del estado de la ligadura correspondiente al primer argumento. Si éste es inactivo se aplica PAR para posibilitar su evaluación en paralelo, con lo que la ligadura activa se liga al segundo argumento del operador `par`, mientras que la ligadura que estaba inactiva pasa a estado ejecutable. En el resto de los casos la regla aplicada es ELIM-PAR, lo que produce como resultado que la ligadura del primer argumento permanezca igual, en tanto que la variable en juego se liga con la expresión que aparece como segundo argumento de `par`.

Observemos que, gracias a la normalización, la generación de ligaduras se realiza solamente al evaluar expresiones `let`, pues el lado izquierdo del operador `par` es una variable que ha de poseer una ligadura en el *heap*. Sin embargo, es únicamente el operador `par` el que introduce paralelismo, pues la declaración local de variables solamente incluirá en el *heap* ligaduras inactivas, en tanto que el operador de paralelo aumenta las ejecutables sin necesidad de que exista demanda sobre ellas. No obstante, en las reglas de paso simple una ligadura en estado ejecutable no puede pasar a ser activa. Para poder realizar esta conversión se emplean las reglas de paso múltiple (sección 5.2.3.2). Además, dicha conversión se realizará de acuerdo a la disponibilidad de recursos por parte del sistema, quedando así patente el hecho de que `par` es sólo una indicación de posible paralelismo, pero no una orden para realizarlo inexorablemente.

5.2.3.2. Transiciones de múltiples pasos

En las transiciones de múltiples pasos es donde el paralelismo de GpH-CORE se hace patente, pues varias ligaduras pueden ser activadas al mismo tiempo. Las reglas que se muestran en la Figura 5.9 se aplican de manera global a todo el *heap* para activar ligaduras, desactivarlas, desbloquearlas y controlar su paralelismo.

$$\frac{H^A = \{x_i \overset{A}{\mapsto} E_i\}_{i=1}^n \quad \{H : x_i \overset{A}{\mapsto} E_i \longrightarrow H_i\}_{i=1}^n}{H \overset{par}{\Longrightarrow} H[\bigcup_{i=1}^n H_i]} \qquad \textbf{(parallel)}$$

$$\frac{\nexists z \overset{B}{\mapsto} E^x \in H}{(H, x \overset{AR}{\mapsto} W, z_1 \overset{B}{\mapsto} E_1^x, \ldots, z_n \overset{B}{\mapsto} E_n^x) \overset{desact}{\longrightarrow} (H, x \overset{I}{\mapsto} W, z_1 \overset{R}{\mapsto} E_1^x, \ldots, z_n \overset{R}{\mapsto} E_n^x)} \qquad \textbf{(deactivation)}$$

$$\frac{|H^A| < \textbf{N° de procesadores} \wedge \ \mathsf{pre}(main, (H + x \overset{A}{\mapsto} E)) \in dom((H + x \overset{A}{\mapsto} E)^A)}{(H, x \overset{R}{\mapsto} E) \overset{act}{\longrightarrow} (H, x \overset{A}{\mapsto} E)} \qquad \textbf{(activation)}$$

$$\frac{H \overset{desact}{\Longrightarrow} K \quad K \overset{act}{\Longrightarrow} H'}{H \overset{sched}{\Longrightarrow} H'} \qquad \textbf{(schedule)}$$

$$\frac{H \overset{par}{\Longrightarrow} K \quad K \overset{sched}{\Longrightarrow} H'}{H \Longrightarrow H'} \qquad \textbf{(compute)}$$

Figura 5.9: GPH-CORE: reglas de transición de paso múltiple

Cómputo paralelo

Es necesario definir cómo las distintas ligaduras se evalúan en paralelo, es decir, cómo se combinan en paralelo los pasos simples de reducción. La regla de paso múltiple que establece la semántica del paralelismo es PARALLEL, que hace evolucionar a todas las ligaduras activas en un *heap*. Tras esta evolución conjunta se obtiene un nuevo *heap*, donde $H[K]$ se define por:

$$H[K] = \{x \overset{\alpha}{\mapsto} E \in H \mid x \notin dom(K)\} \cup K,$$

es decir, se reúnen todas las ligaduras que permanecen invariantes, junto con todas aquéllas que sí han cambiado a consecuencia de la evaluación de las activas.

Desactivación y desbloqueo

En las reglas de paso simple no era posible desbloquear una ligadura. Sin embargo, existen reglas de paso múltiple que sí lo permiten. Además, una ligadura puede pasar de activa o ejecutable a inactiva.

En la regla DESCTIVATION, cuando se ha llegado a una *whnf* en una ligadura cuya variable bloquea otras, la primera pasa a inactiva —pues no se tiene que evaluar más— en tanto que las dependientes pasan de estar bloqueadas a ser ejecutables.

Para conseguir el mayor número de procesadores libres se dispone de *desactivación máxima*, o aplicación reiterada de la desactivación hasta que todas las ligaduras que vinculan a una variable en *whnf* son transformadas en inactivas: $\overset{desact}{\Longrightarrow} = (\overset{desact}{\longrightarrow})^*$, donde en cada paso se reduce el número de ligaduras o activas o ejecutables.

Activación

El paso de una ligadura de ejecutable (R) a activa (A) solamente puede realizarse si el sistema dispone de recursos suficientes, es decir, si existe un procesador libre para albergar a esta ligadura. La regla de pasos múltiples que realiza esta transformación es ACTIVATION.

Como en el caso de la desactivación, la regla de activación se aplicará reiteradamente, $\overset{act}{\Longrightarrow} = (\overset{act}{\longrightarrow})^*$, hasta que se haya convertido la mayor cantidad posible de ligaduras ejecutables en activas, de manera que en cada aplicación el número de ligaduras activas tiene que crecer, asegurando que la ligadura por la que se llegará a la evaluación de *main* tiene precedencia o prioridad sobre el resto. Esto último se logra mediante el predicado sobre pre. La función pre queda definida por

$$\mathsf{pre}(x, K) = \begin{cases} x & si\ x \overset{AR}{\mapsto} E \in K \\ \mathsf{pre}(y, K) & si\ x \overset{B}{\mapsto} E^y \in K \end{cases}$$

donde E^x denota a una expresión *inmediatamente bloqueada* por x, entendiendo por tales aquellas de la forma: $x \mid x\ y \mid x$ `'seq'` E'.

Si no se exigiera el predicado sobre pre, *main* podría estar bloqueada por una secuencia de reducciones terminando en una ligadura ejecutable que podría no ser elegida nunca, con lo que la semántica no expresaría correctamente la terminación de la evaluación. Con pre se asegura que la variable que produce el bloqueo de *main* está siempre activa.

Si se opta por una activación en la que no todas las ligaduras ejecutables se convierten en activas, pero sí que se activan más ligaduras que las meramente demandadas, se introduce no-determinismo, pues no está establecido qué ligadura de entre las ejecutables pasa a ser activa. Sin embargo, a pesar del no-determinismo introducido por tener que escoger qué ligaduras se activan, el paralelismo no hace que el lenguaje pierda la propiedad de confluencia: aunque se pueden realizar distintas reducciones, el valor obtenido por todas ellas como resultado del programa es el mismo. Esta reflexión sobre el no-determinismo solamente tiene sentido cuando el paralelismo explotable en el programa supera al número de procesadores, pues en otro caso no hay que elegir entre todas las ligaduras ejecutables las que se hacen activas.

Planificación

Si se combinan activación y desactivación maximales se obtiene una política de planificación de procesadores con la que se obtiene el máximo paralelismo entre las ligaduras del *heap* con respecto al número de procesadores que posee el sistema. En la regla semántica de planificación primero se lleva a cabo una liberación de procesadores, convirtiendo algunas ligaduras en inactivas, para después transformar en activas todas las que permita el número de procesadores disponibles. La planificación puede ser expresada utilizando la secuenciación:

$$\overset{sched}{\Longrightarrow} = \overset{desact}{\Longrightarrow} ; \overset{act}{\Longrightarrow}$$

donde se aprecia que desactivación y activación se realizan en el orden preciso.

Paso de cómputo

Terminamos especificando cómo se da un paso completo de cómputo en un *heap*: primero evolucionan todas las ligaduras activas y, posteriormente, se realiza una planificación de procesadores para optimizar los recursos. Estos dos pasos dan lugar a la regla COMPUTE de la Figura 5.9.

El orden de aplicación de las reglas, PARALLEL y SCHEDULE es importante, pues si se realizaran en orden inverso podría darse el caso de tener un procesador ocupado en una ligadura activa cuya expresión sea ya una *whnf*.

Empleando la secuenciación, la regla COMPUTE sería:

$$\Longrightarrow = \overset{par}{\Longrightarrow}; \overset{sched}{\Longrightarrow}$$

Una vez vista la definición de la evaluación de un programa en GpH-core, pasamos a tratar las propiedades que Baker-Finch y los demás autores desarrollaron.

5.2.3.3. Propiedades

El interés de la semántica operacional que acabamos de ver es su uso para comparar programas.

Los parámetros para comparar programas se basan en la cantidad de paralelismo que cada uno soporta, el trabajo realizado, la velocidad y la eficiencia. Todos ellos han de extraerse de la secuencia de reducción marcada por la semántica: $H_0 \Longrightarrow H_1 \Longrightarrow \cdots \Longrightarrow H_{t_n}$, donde el número de procesadores es n. Los parámetros son entonces:

Tiempo de ejecución y trabajo realizado: $R(n) = t_n$, $T(n) = \sum_{i=0}^{t_n} |H_i^A|$, es decir, el tiempo de ejecución viene dado por el número de etapas de la secuencia de reducción para n procesadores, mientras que el trabajo realizado se define como el total de ligaduras que han sido activas en algún momento de la ejecución.

Paralelismo medio: $P = \frac{T(\infty)}{t_\infty}$, $P(n) = \frac{T(n)}{t_n}$, o sea, se mide el número de ligaduras activas con respecto al número de etapas que han sido empleadas en la reducción; t_∞ es el número de *heaps* para un número no acotado de procesadores.

Paralelismo máximo: $M = \text{máx}\{|H_i^A|\}_{i=0}^{t_\infty}$, $M(n) = \text{máx}\{|H_i^A|\}_{i=0}^{t_n}$; lo que se mide es el número máximo de ligaduras activas que se podría obtener en un *heap* con respecto al número de procesadores de que se dispone; M corresponde al caso del paralelismo máximo cuando el número de procesadores no está acotado.

Velocidad y eficiencia: $V(n) = \frac{t_1}{t_n}$, $E(n) = \frac{V(n)}{n}$. Estos parámetros se definen en función del número de procesadores (n), obteniéndose mayor velocidad cuantos menos pasos sean necesarios para la obtención del valor del programa, y mayor eficiencia cuanto mayor sea la velocidad y menor el número de procesadores.

Introducidos todos estos parámetros, resulta razonable comparar los programas de dos modos:

- Si los algoritmos son distintos, pueden compararse utilizando el mismo número de procesadores para ver cuál es más eficiente, y, una vez obtenidos los resultados de los parámetros para distinto número de procesadores, se puede ver si la eficiencia cambia al aumentar o disminuir el número de procesadores.
- Dado un único programa, se puede ver como se gana (o se pierde) eficiencia según aumenta el número de procesadores.

Este tipo de comparaciones no busca saber si dos programas son equivalentes, como es el objetivo de otras semánticas como [FH99, Rep93], sino que basa los criterios de comparación en medidas físicas de la ejecución de los programas, como la eficiencia, el número de recursos que emplea o el grado de paralelismo que los programas pueden alcanzar. Este tipo de medidas es fundamental para lenguajes paralelos, pues su objetivo es mejorar la eficiencia de la ejecución.

CAPÍTULO 6

Semántica operacional

Quien todo lo ve, todo lo observa.
Charles-Louis de Secondat
Baron de la Brède et de Montesquieu

La semántica operacional que se presenta en este capítulo define cómo procede la evaluación de una expresión en Jauja (cuya sintaxis se incluyó en la Figura 4.1). Será una descripción a bajo nivel de la ejecución de un programa, pero sin referirnos a ninguna máquina abstracta. La presentación que se realizará será progresiva, introduciéndose por etapas las distintas construcciones de Jauja:

1. λ-cálculo con declaración local de variables y creación de procesos.
2. Incorporación de comunicaciones vía *streams*.
3. Inclusión de no-determinismo explícito en el lenguaje.
4. Creación dinámica de canales de comunicación.

En esta semántica se deben conjugar la evaluación perezosa de construcciones como la aplicación funcional o la declaración local de variables, con la impaciencia a la hora de crear procesos y de evaluar las salidas de los mismos. La evaluación perezosa reduce las expresiones a *whnf*, en tanto que los valores que han de ser comunicados deben ser formas normales, con la salvedad de las abstracciones. Por ello, la semántica operacional

ha de disponer de mecanismos que lleven a cabo estos tipos de evaluación. Denotaremos por *Val* al conjunto de valores *whnf*, que en Jauja restringido estaría compuesto por:

$$W := \lambda x.E \mid \Lambda[x_1 : x_2].E_1[\![E_2 \mid \texttt{nil} \mid [x_1 : x_2]$$

y en adelante denotaremos por $W \in Val$ a los valores *whnf*.

6.1. Un modelo distribuido

Según se acaba de comentar, siguiendo esta semántica se tiene la forma de evaluación precisa de un programa en Jauja. En principio se tiene una única expresión que hay que evaluar. Esta expresión se asociará a una variable inicial, *main*, con consideración especial en el modelo. Sin embargo, la reducción de dicha expresión puede dar lugar a la aparición de más asociaciones, como sucederá en particular al evaluar una declaración `let`, que para evaluar su cuerpo puede necesitar la evaluación de las expresiones ligadas a las variables locales. Por ello se va a considerar que se tiene un conjunto de ligaduras de la forma ⟨variable, expresión⟩, siendo inicialmente dicho conjunto unitario y conteniendo $\langle main, E\rangle$, con E la expresión que representa el programa. No obstante, de esta forma se modelizaría la evaluación de una expresión y de sus subexpresiones locales, pero no se consideraría la existencia de diversos procesos paralelos. Por ello, cuando una variable del conjunto anterior esté ligada a una `#`-expresión se procederá a la creación de un nuevo proceso.

Con el fin de observar el modo en que se comparten las reducciones en este ámbito que conjuga pereza e impaciencia, modelizamos el estado de evaluación mediante un *heap* de ligaduras de expresiones a variables, representando así las clausuras. Bajo el supuesto de que no se permite a los procesos compartir datos, excepto vía comunicación, el *heap* global se distribuye en *heaps* separados, cada uno correspondiente a un proceso del sistema.

Finalmente, no todas las ligaduras se encuentran en la misma situación: unas pueden no haber sido demandadas, otras, habiendo sido demandadas, no pueden evolucionar porque dependen del valor que devolverá otra ligadura, y, por último, otras están evolucionando.

Teniendo en cuenta todas las consideraciones anteriores se procede a definir los sistemas de procesos de la forma siguiente:

Definición 6.1 Sean θ una variable, E una expresión de Jauja y α una etiqueta (A, I o B). Una *ligadura* es una terna $\theta \overset{\alpha}{\mapsto} E$, donde se dice que la variable x está ligada a la expresión E en estado α. Una ligadura puede encontrarse en diferentes *estados de evaluación, cuya etiqueta correspondiente se indica entre paréntesis*:

- Inactiva (I): hasta el momento ninguna otra ligadura ha demandado su evaluación.

- Activa (A): una ligadura en este estado se encuentra en (potencial) ejecución, y no depende, por el momento, de la evaluación de otras variables. Evidentemente, o se trata de la ligadura inicial o ha sido demandada por otra.
- Bloqueada (B): una ligadura se bloquea cuando, estando en ejecución, necesita del valor que se asociará a otra variable, no habiéndose obtenido aún éste.

□

Definición 6.2 *Procesos y sistemas.*

1. Se define un *heap*, H, como un conjunto finito no vacío de ligaduras.
2. Dado un *heap* $H = \{x_1 \overset{\alpha_1}{\mapsto} E_1, \ldots, x_n \overset{\alpha_n}{\mapsto} E_n\}$, el *dominio de* H, $dom(H)$, es el conjunto de las variables en él ligadas, $\{x_1, \ldots, x_n\}$.
3. Un *proceso* es un par $\langle p, H\rangle$, donde p es un identificador de proceso, y H es un *heap*.
4. Un *sistema*, S, es un conjunto no vacío de procesos. Un *sistema finito* es un sistema con una cantidad finita de ligaduras.

Nótese que todo sistema finito contiene un número finito de procesos.

□

Intuitivamente, un sistema representa un conjunto de procesos que se evalúan en paralelo. La evaluación de un programa en Jauja será representada por una secuencia de sistemas, regida por las reglas de transición que introducimos a continuación. Algunas de las ligaduras de un *heap* se ejecutarán en paralelo a su vez, y comparten entre ellas los datos del proceso, es decir, cada una de ellas tendrá acceso a sus vecinas de proceso. Sin embargo, las ligaduras de dos procesos distintos no comparten información, salvo vía comunicaciones.

En lo que sigue, para evitar la repetición de reglas cuya única diferencia es el estado de alguna de las ligaduras, podrá aparecer una ligadura multietiquetada, por ejemplo $\theta \overset{IB}{\mapsto} E$, que indica que la variable θ puede encontrarse en estado inactivo o bloqueado. Por otra parte, la notación $H + \{\theta \overset{\alpha}{\mapsto} E\}$ indica que el conjunto de hebras H se extiende/modifica con la ligadura $\theta \overset{\alpha}{\mapsto} E$; asimismo, $H : \theta \overset{\alpha}{\mapsto} E$ indica que el conjunto H se extiende con la ligadura $\theta \overset{\alpha}{\mapsto} E$ y que es ésta la que guía la regla que se aplica.

Las comunicaciones entre procesos se realizan vía canales, para lo cual se introduce un conjunto especial de identificadores de canal: $\mathcal{C}han$. En nuestro caso, un canal tiene asociado un sentido de comunicación que se mantendrá durante toda su existencia. Tomando el punto de vista del proceso hijo, se tendrá que $\mathcal{C}han = \mathcal{I}n \cup \mathcal{O}ut$, donde $\mathcal{I}n$ son los canales de entrada, es decir, del padre al hijo, mientras que $\mathcal{O}ut$ son los de salida,

i.e. del hijo al padre. Así, en lo que sigue se utilizarán $x, y, z, s, t, u \in \mathit{Var}$ (variables ordinarias de programa), $i \in \mathcal{I}n$, $o \in \mathcal{O}ut$ y $ch \in \mathcal{C}han$, mientras que $\theta \in \mathit{Var} \cup \mathcal{C}han$. Por último, usaremos p y q para representar identificadores de proceso.

Por último introduzcamos una notación que va a aparecer de manera recurrente:

Notación 6.1 *Dado un sistema S, el* proceso principal *es aquel cuyo heap contiene a la variable main.*[1]

□

6.1.1. Sistema de transiciones

De manera similar a como ocurre con otras semánticas operacionales para lenguajes funcionales concurrentes o paralelos [BKT00, FH99, GMP90, PR97], la semántica operacional que aquí se presenta para Jauja tiene dos niveles:

1. Un observador en el *nivel inferior* tendría en su punto de mira la evolución local de cada proceso, que evoluciona independientemente llevando a cabo los cómputos puramente funcionales, como la β-reducción o las ligaduras del `let`. Por lo tanto, en este nivel se van a tener tantos observadores como procesos haya en el sistema.

2. El observador (único) situado en el *nivel superior* controlaría la evolución del sistema en su conjunto, es decir, las interacciones entre los procesos del sistema. Según esto, sus puntos de observación serían la creación de procesos, las comunicaciones entre ellos, etc.

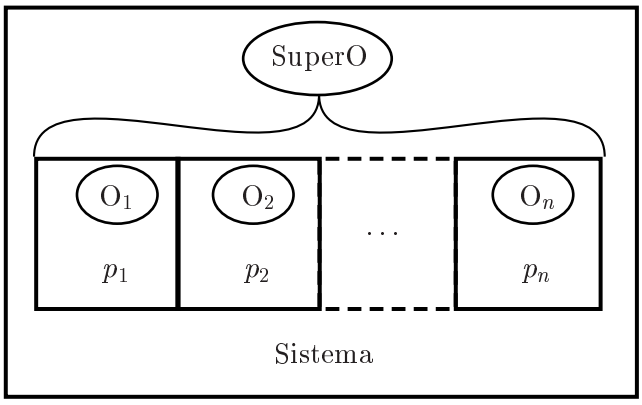


Figura 6.1: Dos niveles de observación

[1] Este proceso es único si la expresión inicial no contiene la variable *main* si ninguno de los renombramientos que empleemos introduce este identificador. De ahora en adelante supondremos ambas condiciones.

Por otra parte, el nivel de abstracción de esta semántica no es tan elevado como el de la semántica natural de Launcbury [Lau93] en la que se consideran pasos *grandes*. A pesar de que la semántica que presentamos aquí considera en último término pasos reiterados, éstos son resultado de la aplicación de varios pasos simples, en el mismo sentido que se emplea en [BKT00].

6.1.1.1. Transiciones locales

Las transiciones locales rigen la evolución individual de cada ligadura, para lo cual tiene que encontrarse activa. Es una evolución local que solamente afecta a ligaduras del mismo proceso que la ligadura en cuestión. Esta actividad interna puede crear, modificar, bloquear o activar clausuras.

6.1.1.2. Transiciones globales

En este caso el observador se encuentra en un peldaño superior y se sitúa en una posición que le permite ver la interacción, mediante comunicaciones, entre los procesos que forman el sistema, así como la creación de los mismos, la gestión de estados (desbloqueo y desactivación de ligaduras) y la planificación de los "procesadores".

El ascenso en la escalera conlleva la realización de varios pasos sencillos (con varios procesos implicados en algunos casos). Así, las reglas globales vendrán definidas por la reiteración de determinadas reglas de paso más pequeño.

Definición 6.3 Sea S un sistema, y $\diamond$ un símbolo que representará el nombre de una regla, para cada regla de paso simple $S \xrightarrow{\diamond} S_1$ definimos una *regla de paso reiterado* $S \overset{\diamond}{\Longrightarrow} S'$ que viene dada por:

1. $S \xrightarrow{\diamond}{}^{*} S'$ y,
2. no existe S" tal que $S' \xrightarrow{\diamond} S''$.

□

Para cada caso particular se demostrará que la aplicación de una regla de paso simple $\diamond$, a una ligadura concreta de un proceso específico, puede hacer posible la aplicación de esa misma regla $\diamond$ a otra ligadura (en el mismo o en otro proceso), pero nunca podrá inhabilitar aplicaciones de $\diamond$ que eran posibles antes de la primera aplicación. Asimismo, para cada caso se demostrará que la imagen por cada regla de paso reiterado está bien definida.

Por último, señalamos que, aunque la presentación de las reglas es paulatina, en cada caso se irán introduciendo ejemplos en los que muchas veces serán necesarios pasos de la semántica que aún no se habrán presentado.

6.2. λ-cálculo y creación de procesos

Comencemos la formalización de esta semántica operacional definiendo la semántica del subconjunto del lenguaje formado por las variables, la abstracción funcional, la aplicación funcional, la aplicación de creación de procesos y la declaración recursiva de variables locales. La intención al elegir este conjunto de construcciones es conformar un cálculo que contenga el núcleo funcional puro de Jauja y la principal característica distintiva del lenguaje, a saber, el paralelismo.

6.2.1. Transiciones locales

El primer grupo de reglas locales, dado en la Figura 6.2, expresa cómo progresa la evaluación perezosa bajo demanda.

$$H + \{x \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} W\} \qquad \textbf{(value)}$$

si $E \notin Val$

$$H + \{x \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\} \qquad \textbf{(demand)}$$

$$H : x \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{B}{\mapsto} x\} \qquad \textbf{(blackhole)}$$

si $E \notin Val$

$$H + \{x \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\, y \longrightarrow H + \{x \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\, y\} \qquad \textbf{(app-demand)}$$

$$H + \{x \overset{I}{\mapsto} \backslash z.E\} : \theta \overset{A}{\mapsto} x\, y \longrightarrow H + \{x \overset{I}{\mapsto} \backslash z.E, \theta \overset{A}{\mapsto} E[y/z]\} \qquad (\beta\textbf{-reduction})$$

$1 \leq i \leq n, fresh(y_i)$

$$H : \theta \overset{A}{\mapsto} \texttt{let}\ \{x_i = E_i\}\ \texttt{in}\ x \longrightarrow \longrightarrow H + \{y_i \overset{I}{\mapsto} E_i[y_1/x_1, \ldots, y_n/x_n]\}_{i=1}^{n} + \{\theta \overset{A}{\mapsto} x[y_1/x_1, \ldots, y_n/x_n]\} \qquad \textbf{(let)}$$

Figura 6.2: Jauja básico: reglas locales

Cuando la evaluación de una expresión finaliza, es decir, ha alcanzado la forma *whnf*, el valor es compartido si otra ligadura lo necesita. Aplicando la regla VALUE se copia y se liga el valor a la variable demandante. No obstante, la demanda puede producirse antes de que haya sido obtenido el valor, bloqueándose la ligadura demandante y activando, si es posible, la demandada. La regla DEMAND es la que lleva a cabo estas modificaciones. Por otra parte, una demanda puede constituir una autorreferencia, no pudiéndose avanzar entonces en la evaluación de la expresión y obligando a un bloqueo de la ligadura en cuestión, lo que queda capturado por la regla BLACKHOLE.

También es objeto de demanda la variable correspondiente a la abstracción cuando se desea realizar una aplicación funcional, ejecutándose esta demanda mediante la regla APP-DEMAND. El traspaso de la abstracción obtenida y su aplicación se realizan mediante la regla β-REDUCTION. Estas reglas difieren de las transiciones locales de [BKT00] por la diferente normalización llevada a cabo en el caso de Jauja.

Finalmente, la declaración local de variables introduce nuevas ligaduras en el *heap* del proceso tratado (regla LET). Todas ellas se etiquetan como inactivas, pues aún no han sido demandadas por el cuerpo de la declaración, que, evidentemente, es la única expresión que las puede demandar. Pero para evitar el choque de nombres es necesario renombrar estas variables locales y, consecuentemente, las expresiones de la declaración. Analicemos el problema con el siguiente ejemplo:

Ejemplo 6.1 *Choque de nombres.*

$$\begin{array}{l}\texttt{let } x_1 = \backslash y.(\texttt{let } z_1 = y,\\ \qquad\qquad\qquad\quad z_2 = (\backslash r.r),\\ \qquad\qquad\qquad\quad z_3 = (z_2)(z_1)\\ \qquad\qquad\quad \texttt{in } z_3),\\ \qquad x_2 = (x_1)(x_3),\\ \qquad x_3 = (x_1)(x_4),\\ \qquad x_4 = 1\\ \texttt{in } x_2\end{array}$$

La aplicación en dos ocasiones de la variable x_1 conlleva tener que evaluar dos veces el `let` interno. Si no se introdujesen nuevos nombres de variables en cada ocasión y se mantuviesen z_1, z_2 y z_3, se tendría más de una ligadura para cada una de estas variables en el *heap* en el que se estuviese evaluando la expresión.

La expresión inicial se liga a la variable especial *main*, y esta ligadura activa es la única presente en el *heap* del sistema inicial.[2]

$$\begin{array}{|l}\hline main \overset{A}{\mapsto} \texttt{let } x_1 = \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3),\\ \qquad\qquad\qquad\quad x_2 = (x_1)(x_3), x_3 = (x_1)(x_4), x_4 = 1\\ \qquad\quad \texttt{in } x_2\\ \hline\end{array}$$

El *heap* va evolucionando como se indica, mediante la aplicación primero de la regla LET, por medio de la cual se introducen las nuevas variables x_{11}, x_{21}, x_{31} y x_{41},

LET

$$\begin{array}{|l}\hline main \overset{A}{\mapsto} x_{21}\\ x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3)\\ x_{21} \overset{I}{\mapsto} (x_{11})(x_{31}),\ x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ x_{41} \overset{I}{\mapsto} 1\\ \hline\end{array}$$

y, a continuación, con la regla DEMAND sobre *main*

DEMAND

$$\begin{array}{|l}\hline main \overset{B}{\mapsto} x_{21}\\ x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3)\\ x_{21} \overset{A}{\mapsto} (x_{11})(x_{31}),\ x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ x_{41} \overset{I}{\mapsto} 1\\ \hline\end{array}$$

La variable x_{11} ya tiene ligada una abstracción, por lo que se puede llevar a cabo la β-reducción correspondiente:

[2] En los ejemplos, cada *heap* se representa con un marco con las ligaduras en su interior, siendo el color de las ligaduras azul para las inactivas, verde para las activas y rojo para las bloqueadas.

β-REDUCTION

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{A}{\mapsto} \texttt{let } z_1 = x_{31}, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3 \\
x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ \ x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

y a continuación LET sobre x_{21}:

LET

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{A}{\mapsto} z_{31} \\
z_{11} \overset{I}{\mapsto} x_{31},\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{31} \overset{I}{\mapsto} (z_{21})(z_{11}) \\
x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ \ x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

En este punto la regla LET produce la introducción de z_1, z_2 y z_3 por vez primera en el *heap* (como z_{11}, z_{21} y z_{31}).

DEMAND

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{B}{\mapsto} z_{31} \\
z_{11} \overset{I}{\mapsto} x_{31},\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{31} \overset{A}{\mapsto} (z_{21})(z_{11}) \\
x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ \ x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

β-REDUCTION

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{B}{\mapsto} z_{31} \\
z_{11} \overset{I}{\mapsto} x_{31},\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{31} \overset{A}{\mapsto} z_{11} \\
x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ \ x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

DEMAND

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{B}{\mapsto} z_{31} \\
z_{11} \overset{A}{\mapsto} x_{31},\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{31} \overset{B}{\mapsto} z_{11} \\
x_{31} \overset{I}{\mapsto} (x_{11})(x_{41}),\ \ x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

DEMAND

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{B}{\mapsto} z_{31} \\
z_{11} \overset{B}{\mapsto} x_{31},\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r) \\
z_{31} \overset{B}{\mapsto} z_{11},\ \ x_{31} \overset{A}{\mapsto} (x_{11})(x_{41}),\ \ x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

β-REDUCTION

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{21} \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{B}{\mapsto} z_{31} \\
z_{11} \overset{B}{\mapsto} x_{31},\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{31} \overset{B}{\mapsto} z_{11} \\
x_{31} \overset{A}{\mapsto} \texttt{let } z_1 = x_{41}, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3 \\
x_{41} \overset{I}{\mapsto} 1
\end{array}
$$

La regla LET nos lleva a introducir z_1, z_2 y z_3 una segunda vez (como z_{12}, z_{22} y z_{32}). Si no se hubiera realizado un renombramiento en esta regla se produciría el choque de nombres entre la incorporación previa y ésta. En este ejemplo también se ha observado que en las abstracciones no es necesario renombrar, pues la β-reducción sustituye la variable ligada (parámetro formal) por la variable argumento (parámetro actual). No detallamos todos los pasos siguientes de la reducción, pero al final se tendría:

$$
\begin{array}{l}
main \overset{I}{\mapsto} 1 \\
x_{11} \overset{I}{\mapsto} \backslash y.(\texttt{let } z_1 = y, z_2 = (\backslash r.r), z_3 = (z_2)(z_1) \texttt{ in } z_3) \\
x_{21} \overset{I}{\mapsto} 1 \\
z_{11} \overset{I}{\mapsto} 1,\ \ z_{21} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{31} \overset{I}{\mapsto} 1 \\
x_{31} \overset{I}{\mapsto} 1 \\
z_{12} \overset{I}{\mapsto} 1,\ \ z_{22} \overset{I}{\mapsto} (\backslash r.r),\ \ z_{32} \overset{I}{\mapsto} 1 \\
x_4 \overset{I}{\mapsto} 1
\end{array}
$$

□

Conviene observar que ni la variable de la abstracción ni la del argumento de una aplicación podrán ser canales de comunicación, debido a que la normalización introduce variables intermedias en caso de no que se tratara ya de variables. Del mismo modo, no pueden aparecer autorreferencias de canales, pues, como se verá más adelante en la Sección 6.2.2.1, una variable en un *heap* no puede ligarse a un canal del mismo proceso, i.e. $\theta \overset{\alpha}{\mapsto} ch \in H \Longrightarrow ch \notin dom(H)$.

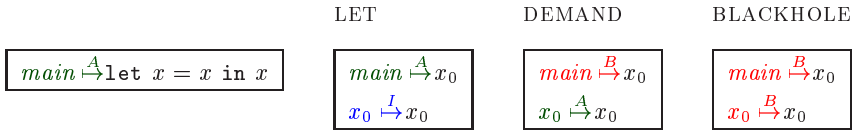
Las reglas descritas hasta este momento marcan la evolución de un proceso en su interior siempre que no tenga que interactuar con el resto de procesos del sistema. En la siguiente Sección (6.2.2) se tratan las transiciones que involucran a más de un proceso, así como las tareas que tiene que realizar el planificador del sistema. Pero veamos previamente cómo las reglas expuestas detectan autorreferencias e interbloqueos.

Ejemplo 6.2 *Agujero negro directo: autorreferencia.*

Si consideramos la expresión ya normalizada

$$\texttt{let } x = x \texttt{ in } x$$

en la que aparece una autorreferencia de la variable x, su evaluación dará lugar a:



donde observamos que la regla BLACKHOLE detecta la autorreferencia de la variable x_0 (que es el renombramiento de x introducido por la regla LET).

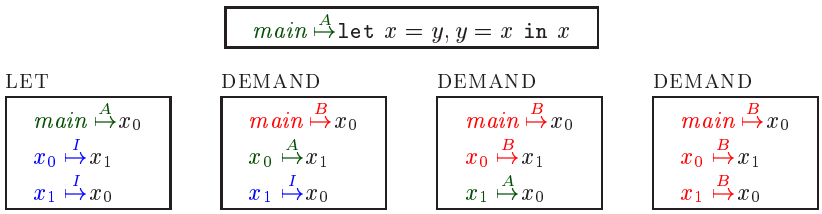
□

Ejemplo 6.3 *Agujero negro indirecto: interbloqueo.*

En la expresión:

$$\texttt{let } x = y, y = x \texttt{ in } x$$

la variable x se define como y, que, a su vez, es definida como x. La aplicación del sistema de transiciones locales ofrece como resultado:



la reiterada aplicación de la regla DEMAND solicitando la evaluación de variables deja bloqueadas todas las ligaduras del *heap*, con lo que el interbloqueo ha sido detectado.

□

Finalmente, veamos en el siguiente ejemplo el funcionamiento de la regla APP-DEMAND, similar a la regla DEMAND:

Ejemplo 6.4 *Aplicación funcional con demanda.*

$$((\backslash x.x)(\backslash x.x))\,1$$

Tras normalizar esta expresión se obtiene:

$$\begin{array}{l}\texttt{let } x_5 = 1,\\ \quad x_6 = \texttt{let } x_1 = (\backslash x.x),\\ \qquad\qquad\quad x_2 = (\backslash x.x),\\ \qquad\qquad\quad x_3 = (x_2)(x_1)\\ \qquad\quad \texttt{in } x_3,\\ \quad x_7 = (x_6)(x_5)\\ \texttt{in } x_7\end{array}$$

y su evaluación procede de la forma siguiente:

$$main \overset{A}{\mapsto} \texttt{let } x_5 = 1, x_6 = \texttt{let } x_1 = (\backslash x.x), x_2 = (\backslash x.x), x_3 = (x_2)(x_1) \texttt{ in } x_3, x_7 = (x_6)(x_5) \texttt{ in } x_7$$

LET

$$\begin{array}{l} main \overset{A}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{I}{\mapsto} \texttt{let } x_1 = (\backslash x.x), x_2 = (\backslash x.x), x_3 = (x_2)(x_1) \texttt{ in } x_3 \\ x_{10} \overset{I}{\mapsto} (x_9)(x_8) \end{array}$$

DEMAND

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{I}{\mapsto} \texttt{let } x_1 = (\backslash x.x), x_2 = (\backslash x.x), x_3 = (x_2)(x_1) \texttt{ in } x_3 \\ x_{10} \overset{A}{\mapsto} (x_9)(x_8) \end{array}$$

En el siguiente paso no se puede llevar a cabo la β-reducción $x_9\, x_8$ porque la variable x_9 no se encuentra ligada a una λ-abstracción. Para poder proseguir la evaluación se efectúa una demanda para evaluar dicha variable:

APP-DEMAND

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{A}{\mapsto} \texttt{let } x_1 = (\backslash x.x), x_2 = (\backslash x.x), x_3 = (x_2)(x_1) \texttt{ in } x_3 \\ x_{10} \overset{B}{\mapsto} (x_9)(x_8) \end{array}$$

LET

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{A}{\mapsto} x_{13} \\ x_{10} \overset{B}{\mapsto} (x_9)(x_8) \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{I}{\mapsto} (x_{12})(x_{11}) \end{array}$$

DEMAND

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{B}{\mapsto} x_{13} \\ x_{10} \overset{B}{\mapsto} (x_9)(x_8) \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{A}{\mapsto} (x_{12})(x_{11}) \end{array}$$

β-REDUCTION

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{B}{\mapsto} x_{13} \\ x_{10} \overset{B}{\mapsto} (x_9)(x_8) \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{A}{\mapsto} x_{11} \end{array}$$

VALUE

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{B}{\mapsto} x_{13} \\ x_{10} \overset{B}{\mapsto} (x_9)(x_8) \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{A}{\mapsto} (\backslash x.x) \end{array}$$

VALUE

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{A}{\mapsto} (\backslash x.\ x) \\ x_{10} \overset{B}{\mapsto} (x_9)(x_8) \\ x_{11} \overset{I}{\mapsto} (\backslash x.\ x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.\ x) \\ x_{13} \overset{I}{\mapsto} (\backslash x.\ x) \end{array}$$

β-REDUCTION

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{I}{\mapsto} (\backslash x.x) \\ x_{10} \overset{A}{\mapsto} x_8 \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{I}{\mapsto} (\backslash x.x) \end{array}$$

VALUE

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{I}{\mapsto} (\backslash x.x) \\ x_{10} \overset{A}{\mapsto} 1 \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{I}{\mapsto} (\backslash x.x) \end{array}$$

VALUE

$$\begin{array}{l} main \overset{A}{\mapsto} 1 \\ x_8 \overset{I}{\mapsto} 1 \\ x_9 \overset{I}{\mapsto} (\backslash x.x) \\ x_{10} \overset{I}{\mapsto} 1 \\ x_{11} \overset{I}{\mapsto} (\backslash x.x) \\ x_{12} \overset{I}{\mapsto} (\backslash x.x) \\ x_{13} \overset{I}{\mapsto} (\backslash x.x) \end{array}$$

Se observa cómo se reducen las aplicaciones funcionales, demandando primero la abstracción para, una vez obtenida, proceder a la aplicación. Se han saltado los pasos del planificador, que se detallarán en la Sección 6.2.2, que comprenden la desactivación y el desbloqueo de ligaduras, por ejemplo entre las aplicación de VALUE y de β-REDUCTION. □

6.2.2. Transiciones globales

Los cambios del sistema que puede atalayar nuestro observador del peldaño superior son la creación de procesos y la comunicación entre ellos. Además, también son reglas globales las que desbloquean y desactivan ligaduras, pues éstas son las tareas que competen al planificador.

6.2.2.1. Creación de procesos

Un proceso se crea en presencia de una #-expresión, y siempre que:

1. El cuerpo no dependa de un valor que aún no haya sido comunicado por un canal,
2. Toda dependencia libre del cuerpo esté ligada a un valor *whnf*.

Una *dependencia libre* se define como el cierre por transitividad del concepto de variable libre, es decir, si x es una variable libre de E, x está ligada a E' e y es una variable libre de E', entonces y es dependencia libre de E. Asimismo, también son dependencias libres las derivadas de y.

La creación de procesos se rige por la regla de la Figura 6.3.

$$\textbf{(process creation)}$$
$$\text{si } \mathsf{nf}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}) = \emptyset$$
$$\mathit{fresh}(q, z, i, o), \mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}\rangle) \xrightarrow{pc} (S, \langle p, H + \{\theta \overset{B}{\mapsto} o, i \overset{A}{\mapsto} y\}\rangle, \langle q, \eta(\mathtt{nh}(x, H)) + \{o \overset{A}{\mapsto} \eta(x)\, z, z \overset{B}{\mapsto} i\}\rangle)$$

Figura 6.3: Jauja básico: creación de procesos

Como se explicó en el Capítulo 4, cuando una #-expresión se encuentra a nivel superior, es decir, directamente ligada a una variable del *heap*, se procede a la creación del correspondiente proceso. De este modo se genera paralelismo especulativo, aplicando la regla anterior incluso a ligaduras inactivas, es decir, no demandadas.

En la regla PROCESS CREATION se determina qué partes de la expresión de creación van a ser evaluadas por cada uno de los procesos involucrados. Así, tanto el cuerpo del proceso, x, como su argumento, y, serán evaluados por el padre (p). Por último, la aplicación, $x\, y$, será evaluada por el hijo recién nacido (q), que, en caso de necesitar el

valor a que y dé lugar, deberá esperar a que éste le sea comunicado. Y mientras el padre espera a que el hijo le comunique valor resultante de la aplicación, la ligadura que en él estaba ligada a la #-expresión se bloquea en un nuevo canal de salida $o \in \mathcal{O}ut$, en tanto que en el hijo se introduce una variable cuya ligadura queda bloqueada en el canal de entrada $i \in \mathcal{I}n$.

En el Capítulo 4 también se mencionó la existencia de una aplicación impaciente. Este es el punto de la evaluación donde se materializa este tipo de la evaluación: los canales creados como consecuencia del nacimiento del nuevo proceso se consideran ligaduras activas, y se evaluarán si se dispone de suficientes recursos.

En cuanto a dependencias, por una parte, la evaluación de un proceso necesitará la presencia en su *heap* de determinados valores. Ahora bien, en principio dichos valores solamente pueden ser comunicados vía canales, y no existe un *heap* común accesible por todos los procesos. Sin embargo, cuando un proceso es creado, nace con un *heap* inicial que contiene todas las ligaduras de las que puede depender el cuerpo del proceso, i.e. sus dependencias libres; asimismo, cuando se comunica un valor se copiarán en el *heap* del receptor todas las ligaduras correspondientes a las dependencias libres de dicho valor. En los dos casos, la copia solamente se puede realizar si dichas variables están ligadas a valores en *whnf*, por lo que la creación de un proceso (y también la comunicación, como veremos más adelante), no se llevará a cabo si no se verifica esta condición. En la regla en estudio, dicha condición viene expresada por: $\mathsf{nf}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}) = \emptyset$ (las dependencias libres necesarias no evaluadas bloquean).

Para poder recolectar las ligaduras se dispone de la función $\mathsf{nh}(E, H)$ (*heap* necesario) que colecta del *heap* H las ligaduras que E podría necesitar en su evaluación. La incorporación de estas ligaduras al *heap* del correspondiente proceso se hace después de someterlas a un renombramiento con variables frescas (η), para que no se produzcan choques de nombres.

Por otra parte, como se ha mencionado arriba, la creación de un proceso se bloquea si se está en espera de valores que han de ser comunicados vía canales, o incluso a la creación de otro proceso. Este es un caso particular de dependencia de una variable que no está ligada a una expresión en *whnf*, por lo que su detección queda resuelta por la función nf (libres (*free*) necesarias) que recolecta todas las dependencias debidas a variables libres. Puede darse el caso, como veremos en el Ejemplo 6.8, de que la creación esté esperando a que ella misma se cree, con lo que se producirá un interbloqueo permanente de dicha creación.

Ambas funciones quedan recogidas en la Figura 6.4, donde se incluye la función auxiliar dexp, que dada una expresión devuelve sus subexpresiones, o expresiones de las que *depende*, salvo en el caso de variables, en el que devuelve el conjunto vacío. Se emplea también la notación siguiente:

Notación 6.2 Llamamos *expresiones inmediatamente bloqueadas en la variable* x, y las denotamos por E^x_B, a aquellas de la forma: x o bien $x\,y$, donde y es una variable

cualquiera.

□

Es decir, E^x_B indica que la expresión E para proseguir su evaluación necesita que la variable x esté evaluada, y que esta expresión es de la forma $y \,|\, y\, z$, como se deduce de las reglas locales DEMAND y APP-DEMAND.

$$\begin{aligned}
\mathtt{dexp}(\theta) &= \emptyset \\
\mathtt{dexp}(\backslash x.E) &= \{E\} \\
\mathtt{dexp}(E_1\ E_2) &= \{E_1, E_2\} \\
\mathtt{dexp}(E_1 \# E_2) &= \{E_1, E_2\} \\
\mathtt{dexp}(\mathtt{let}\ \{x_i = E_i\}_n\ \mathtt{in}\ E) &= \{E_1, \ldots, E_n, E\}
\end{aligned}$$

$$\mathsf{nh}(E,H) \models \{\text{(i)},\text{(ii)}\} \wedge \forall A.(A \subseteq H \wedge A \models \{\text{(i)},\text{(ii)}\} \Rightarrow \mathsf{nh}(E,H) \subseteq A)$$

$$\begin{aligned}
&\text{(i)} \quad x \overset{\alpha}{\mapsto} E \in H \Rightarrow \{x \overset{I}{\mapsto} E\} \cup \mathsf{nh}(E,H) \subseteq \mathsf{nh}(x,H) \\
&\text{(ii)} \quad \bigcup_{E' \in \mathtt{dexp}(E)} \mathsf{nh}(E',H) \subseteq \mathsf{nh}(E,H)
\end{aligned}$$

$$\mathsf{nf}(E,H) \models \{\text{(i)-(v)}\} \wedge \forall A.(A \subseteq H \wedge A \models \{\text{(i)-(v)}\} \Rightarrow \mathsf{nf}(E,H) \subseteq A)$$

$$\begin{aligned}
&\text{(i)} \quad x \overset{\alpha}{\mapsto} W \in H \ \Rightarrow\ \mathsf{nf}(W,H) \subseteq \mathsf{nf}(x,H) \\
&\text{(ii)} \quad x \overset{IA}{\mapsto} E \in H \wedge E \notin \mathit{Val} \ \Rightarrow\ \{x \overset{IA}{\mapsto} E\} \subseteq \mathsf{nf}(x,H) \\
&\text{(iii)} \quad x \overset{B}{\mapsto} E \in H \wedge E \notin \mathit{Val} \wedge (E \equiv E^y_B \vee E \equiv y\#z) \ \Rightarrow\ \{x \overset{B}{\mapsto} E\} \cup \mathsf{nf}(y,H) \subseteq \mathsf{nf}(x,H) \\
&\text{(iv)} \quad x \overset{B}{\mapsto} E \in H \wedge E \notin \mathit{Val} \wedge E \equiv \mathit{ch} \ \Rightarrow\ \{x \overset{B}{\mapsto} E\} \subseteq \mathsf{nf}(x,H) \\
&\text{(v)} \quad \bigcup_{E' \in \mathtt{dexp}(E)} \mathsf{nf}(E',H) \subseteq \mathsf{nf}(E,H)
\end{aligned}$$

Figura 6.4: Jauja básico: detección de dependencias y recolección de ligaduras

Las funciones nh y nf se construyen como el menor subconjunto de ligaduras de H que satisfacen las propiedades indicadas para cada una. Si P es un conjunto de propiedades, denotaremos por $A \models P$ el hecho de que A satisface todas las propiedades de P. En ambas funciones, a las que denominaremos de manera genérica f, $f(E,H)$ es el menor subconjunto de H que verifica las propiedades indicadas, cuando se va tratando E de manera estructural.

Recolección de ligaduras, $\mathsf{nh}(E,H)$*:* si la expresión E es una variable, la ligadura de dicha variable en el *heap* H debe estar presente en $\mathsf{nh}(E,H)$, aunque inactiva, pues en el proceso en el que se copiará posteriormente este conjunto (de ligaduras) la ligadura copiada todavía no ha sido demandada. Además, la recolección se propaga a la expresión ligada a la variable, cuya recolección de ligaduras también ha de estar incluida en $\mathsf{nh}(E,H)$. Si E es cualquier otra expresión, se procede a la recolección de ligaduras de todas sus subexpresiones, o expresiones dependientes ($\mathtt{dexp}(E)$).

Detección de dependencias de libres, $\mathsf{nf}(E,H)$*:* en caso de que E sea una variable, x, caben dos posibilidades: (1) que x esté ligada a un valor *whnf*, o (2) que x esté ligada

a otra expresión. En el primer caso se han de recoger las dependencias libres derivadas del valor; en el segundo, caben de nuevo dos posibilidades: (a) que la ligadura de x sea inactiva o activa, lo cual hace que esta ligadura deba estar incluida entre las dependencias libres de x, o (b) que la ligadura de x esté bloqueada. Esta situación se puede dar por cuatro razones:

1. Se bloqueó por la aplicación de DEMAND, por lo que está bloqueada en otra variable.
2. El motivo del bloqueo fue la aplicación de APP-DEMAND, y, por tanto, está bloqueada en una aplicación.
3. Se encuentra ligada a una creación de proceso que no pudo realizarse porque la variable de la abstracción tiene a su vez dependencias sin resolver.
4. La variable está bloqueada en un canal, producto de las modificaciones del sistema derivadas de una creación de proceso.

En todos los casos es evidente cuál es la variable que hace que existan dependencias libres, y o ch. En consecuencia, han de estar entre las dependencias libres de x la propia ligadura de x y las dependencias libres derivadas de y o ch, según corresponda.

Finalmente, si E no es una variable, se recurre a todas sus subexpresiones para localizar las dependencias libres de E.

Dado un sistema, la regla de creación de procesos $\xrightarrow{pc}$ se aplica reiteradamente sobre dicho sistema hasta que ninguna creación es posible. Este hecho queda recogido en la regla de paso reiterado

$$\stackrel{pc}{\Longrightarrow}.$$

En esta regla no se predetermina ningún orden entre las creaciones de proceso posibles. Sin embargo, esto no representa ningún obstáculo para la correcta evaluación del programa, pues la realización de una creación de proceso nunca habilita una nueva creación, y tampoco imposibilita la creación de un proceso que antes era posible crear. Para demostrar estos resultados, sinteticemos las ocasiones en las que se puede realizar una creación de proceso:

Definición 6.4 Sean H un *heap* y $\theta \overset{\alpha}{\mapsto} x\#y$ una ligadura de H, $x\#y$ es una *creación de proceso factible* si $\mathsf{nf}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}) = \emptyset$.

□

Demostremos ahora los resultados que se han descrito informalmente antes.

Proposición 6.1 *Sean S un sistema, H un heap de S, y $\theta_1 \overset{\alpha_1}{\mapsto} x_1\#y_1 \in H$ una creación factible. Sea $\theta_2 \overset{\alpha_2}{\mapsto} x_2\#y_2 \in H$ una creación no factible porque $\theta_1 \overset{\alpha_1}{\mapsto} x_1\#y_1 \in \mathsf{nf}(x_2, H)$. Después de crear $x_1\#y_1$, la creación $x_2\#y_2$ sigue siendo no factible.*

Demostración P.6.1
Si $\theta_1 \overset{\alpha_1}{\mapsto} x_1\#y_1 \in \mathsf{nf}(x_2, H)$ es porque es necesaria la evaluación de θ_1 para crear el proceso de θ_2. Después de crear el proceso de θ_1, su ligadura es $\theta_1 \overset{B}{\mapsto} o_1$, con lo que tampoco está evaluada y $\theta_1 \overset{B}{\mapsto} o_1 \in \mathsf{nf}(x_2, H)$. En consecuencia, $x_2\#y_2$ no puede ser creado.

c.q.d.

Proposición 6.2 *Sean S un sistema, H_1 un heap de S, y $\theta_1 \overset{\alpha_1}{\mapsto} x_1\#y_1 \in H_1$ una creación factible. Sean H_2 un heap de S, posiblemente el propio H_1, y $\theta_2 \overset{\alpha_2}{\mapsto} x_2\#y_2 \in H_2$ una creación también factible tal que $\theta_1 \neq \theta_2$. Después de crear $x_1\#y_1$, $x_2\#y_2$ sigue siendo factible.*

Demostración P.6.2
La condición para que $x_2\#y_2$ pueda crearse es que $\mathsf{nf}(x_2, H_2) = \emptyset$. Tenemos que probar que después de crear $x_1\#y_1$ este conjunto sigue siendo vacío.

Si $H_1 \neq H_2$ es evidente, pues la creación de $x_1\#y_1$ en H_1 no modifica el *heap* H_2, solamente H_1 y el *heap* del nuevo proceso recién creado.

En el caso de que $H_1 = H_2$, después de crear $x_1\#y_1$, H_1 se transforma en $H_1' = H_1 + \{\theta_1 \overset{B}{\mapsto} o_1, i_1 \overset{A}{\mapsto} y_1\}$. Si antes de la creación $\mathsf{nf}(x_2, H_2) = \emptyset$, x_2 no dependía de θ_1; ahora no se ha modificado ninguna ligadura salvo esta, con lo que tampoco puede depender de θ_1. La ligadura nueva es la del canal $i_1 \overset{A}{\mapsto} y_1$, al ser i_1 un canal nuevo y que solamente aparece en esta ligadura dentro del *heap* H_1', no puede ser que x_2 dependa de él. Por lo tanto $\mathsf{nf}(x_2, H_1')$ sigue siendo vacío y la creación $x_2\#y_2$ sigue siendo factible.

c.q.d.

Teorema 6.1 *Sea S un sistema, y sean $\theta_1 \overset{\alpha_1}{\mapsto} x_1\#y_1 \in H_1, \ldots, \theta_n \overset{\alpha_n}{\mapsto} x_n\#y_n \in H_n$ todas las ligaduras de creaciones factibles de S. Después de aplicar $\overset{pc}{\Longrightarrow}$ justo han sido realizadas estas creaciones.*

Demostración T.6.1
Por la Proposición 6.2 la realización de una cualquiera de ellas no impide que ninguna de las restantes pueda ser realizada después, por lo que todas ellas se habrán realizado tras aplicar $\overset{pc}{\Longrightarrow}$. Por la Proposición 6.1 ninguna de ellas habilita la creación de un proceso cuya ligadura no esté en la lista, por lo que tras la aplicación de $\overset{pc}{\Longrightarrow}$ solamente se habrán creado los procesos de las ligaduras de la lista.

Además, es necesario que la regla $\overset{pc}{\Longrightarrow}$ esté bien definida.

Proposición 6.3 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{pc}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{pc}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.3
En la Sección A.1.

Como veremos al final de esta sección, cuando hayamos tratado todas las reglas de paso reiterado, si se parte de un sistema finito la incorporación de las nuevas reglas lo mantienen finito. Por lo tanto, a lo largo de nuestras evaluaciones siempre manejaremos sistemas finitos, y, como consecuencia de la Proposición 6.3, la aplicación de $\overset{pc}{\Longrightarrow}$ siempre termina. Informalmente, al partir de un sistema finito todas las reglas de paso reiterado veremos que están bien definidas. Las reglas globales simples que las componen posiblemente incrementarán el tamaño del sistema, pero siempre de manera finita. En consecuencia, la finitud de los sistemas se mantendrá.

Veamos ejemplos del funcionamiento de la creación de procesos. En ellos, un proceso se representa de la forma:

nombre_proceso (N° Hijos: i)
$variable_1 \overset{\alpha_1}{\mapsto} E_1$
...
$variable_n \overset{\alpha_n}{\mapsto} E_n$

Ejemplo 6.5 *Creación de proceso factible.*

Observemos la evaluación de la siguiente creación de proceso:

$$(\backslash x.x)\#(1)$$

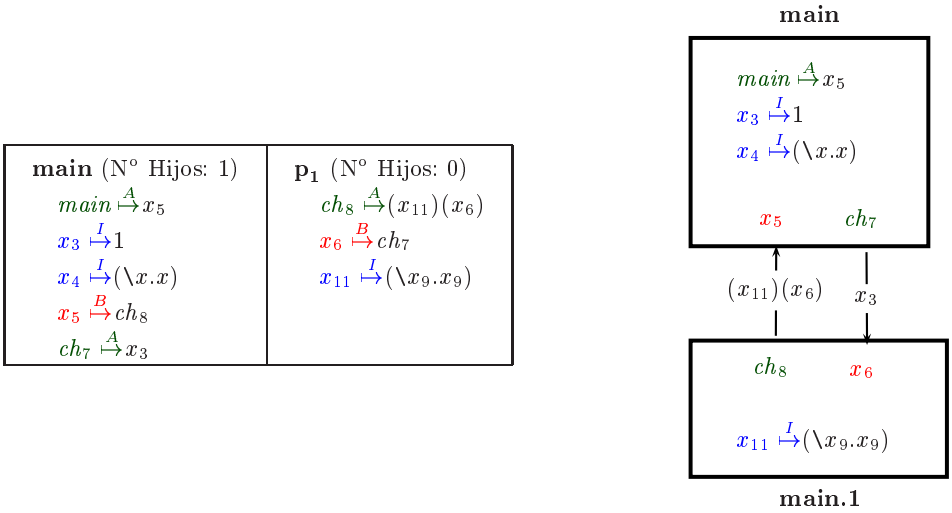
Tras normalizar la expresión e incorporarla como expresión principal se obtiene el proceso:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x_0 = (\backslash x.x), x_1 = 1, x_2 = (x_0)\#(x_1), \texttt{ in } x_2$

El primer paso de la evaluación es la aplicación de la regla local LET.

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_5$
$x_3 \overset{I}{\mapsto} 1$
$x_4 \overset{I}{\mapsto} (\backslash x.x)$
$x_5 \overset{I}{\mapsto} (x_4)\#(x_3)$

Analizando el conjunto $\mathsf{nf}(x_4, H_{main})$ vemos que se trata del conjunto vacío, pues x_4 está ligada a un valor *whnf* que no contiene variables libres. Esto hace que se pueda realizar la creación de proceso.



El canal de entrada al hijo (p_1) es ch_7, y la salida de p_1 es ch_8. La ligadura que se ha copiado del padre al hijo ha sido $x_4 \overset{I}{\mapsto} \backslash x.x$, pero renombrada como $x_{11} \overset{I}{\mapsto} \backslash x_9.x_9$, por si se tuviera que copiar en otra ocasión (lo que sucedería, por ejemplo, si x_4 fuera una variable libre de un valor a comunicar del proceso principal, main, a p_1, o si el valor ligado a x_{11} tuviera que ser enviado de p_1 al proceso main).

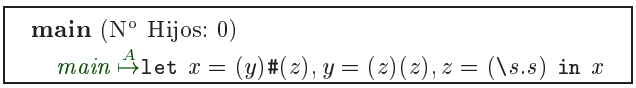
□

Veamos dos ejemplos en los que la creación del nuevo proceso queda bloqueada en espera de que las variables de las que depende la #-expresión sean evaluadas.

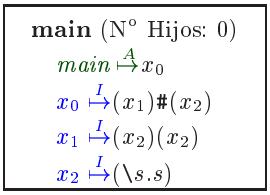
Ejemplo 6.6 *Creación de proceso no factible: dependencia directa.*

Consideremos la expresión:

$$\texttt{let } x = (y)\#(z), y = (z)(z), z = (\backslash s.s) \texttt{ in } x$$



El primer paso consiste en ejecutar la regla LET para incorporar las nuevas ligaduras en el *heap*:



Observamos que la creación de proceso ligada a x_0 depende de la variable x_1 que se encuentra sin evaluar, por lo que todavía no se puede crear el nuevo proceso. En el Ejemplo 6.15, y tras haber incorporado las transiciones globales restantes, se mostrará cómo

se procede ante la imposibilidad de crear el proceso.

□

Ejemplo 6.7 *Creación de proceso no factible: dependencia indirecta.*

Veamos la siguiente expresión, ya normalizada:

$$\texttt{let } x = (y)\#(z), y = (\backslash s.z), z = (t)(t), t = (\backslash u.u) \texttt{ in } x$$

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = (y)\#(z), y = (\backslash s.z), z = (t)(t), t = (\backslash u.u) \texttt{ in } x$

Al igual que en el Ejemplo 6.6, el primer paso consiste en ejecutar la regla LET para incorporar las nuevas ligaduras en el *heap*:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} (x_1)\#(x_2)$
$x_1 \overset{I}{\mapsto} (\backslash s.(x_2))$
$x_2 \overset{I}{\mapsto} (x_3)(x_3)$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$

En este caso, en la creación $x_1 \# x_2$ la variable x_1 está ligada a un valor *whnf*. Sin embargo, la dependencia de libres $\mathsf{nf}(x_1, \text{main})$ devuelve $\{x_2 \overset{I}{\mapsto} x_3\, x_3\}$. Al ser no vacío este conjunto, no se puede proceder con la creación. Pero la dependencia no es directa: x_1 no está en este conjunto de ligaduras, sino que lo que tenemos es una variable libre en la expresión ligada a x_1.

□

Ejemplo 6.8 *Creación auto-bloqueada.*

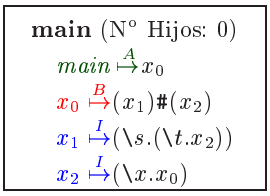
$$\texttt{let } z_1 = (z_2)\#(z_3), z_2 = (\backslash s.(\backslash t.z_3)), z_3 = (\backslash x.z_1) \texttt{ in } z_1$$

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } z_1 = (z_2)\#(z_3), z_2 = (\backslash s.(\backslash t.z_3)), z_3 = (\backslash x.z_1) \texttt{ in } z_1$

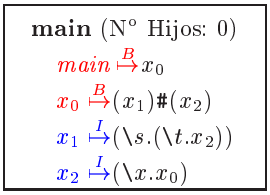
Se aplica la regla LET obteniendo:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} (x_1)\#(x_2)$
$x_1 \overset{I}{\mapsto} (\backslash s.(\backslash t.x_2))$
$x_2 \overset{I}{\mapsto} (\backslash x.x_0)$

La creación del proceso ligado a x_0 no puede realizarse porque $\mathsf{nf}(x_1, \text{main}) = \{x_0 \overset{I}{\mapsto} (x_1)\#(x_2)\}$, es decir, existen dependencias libres sin evaluar.



La variable *main*, tras aplicar la regla DEMAND, se bloquea porque espera la evaluación de x_0, que a su vez está bloqueada.

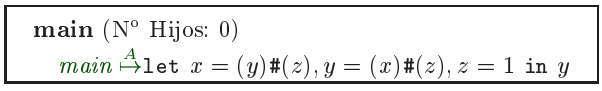


Este ejemplo ha mostrado otra forma de interbloqueo.

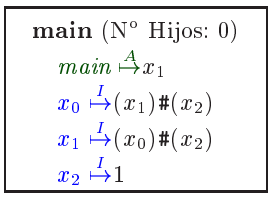
□

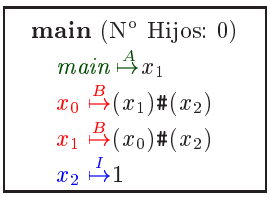
Ejemplo 6.9 *Interdependencia entre creaciones de proceso.*

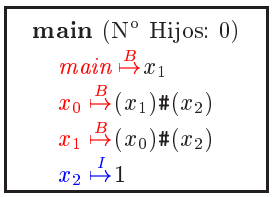
$$\texttt{let } x = (y)\#(z), y = (x)\#(z), z = 1 \texttt{ in } y$$



En esta expresión la creación del proceso de x depende de la creación del proceso correspondiente a y, y viceversa.







Tras la evaluación local mediante LET, procedería la creación de los procesos de nivel superior. La interdependencia entre las dos ligaduras hace imposible creación alguna. La regla global de bloqueo de creaciones de proceso, que introduciremos y estudiaremos más adelante, bloquea las ligaduras de creación no realizadas, dando como resultado el sistema final, donde ninguna ligadura se encuentra activa, por lo que la evaluación se detiene.

□

Ejemplo 6.10 *Demora en la creación de un proceso: esperando una comunicación.*

$$\texttt{let } x = (y)\#(z), y = (s)\#(z), s = (\backslash t.(\backslash t'.t)), z = 1 \texttt{ in } y$$

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = (y)\#(z), y = (s)\#(z), s = (\backslash t.(\backslash t'.t)), z = 1 \texttt{ in } y$

Se añaden las nuevas variables locales en el *heap*:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_1$
$x_0 \overset{I}{\mapsto} (x_1)\#(x_3)$
$x_1 \overset{I}{\mapsto} (x_2)\#(x_3)$
$x_2 \overset{I}{\mapsto} (\backslash t.(\backslash t'.t))$
$x_3 \overset{I}{\mapsto} 1$

El proceso ligado a x_1 puede crearse, pero el ligado a x_0 aún no, por estar esperando a que x_1 esté ligada a un valor *whnf*. Con la aplicación de $\overset{pc}{\Longrightarrow}$ se crea dicho proceso ligado a x_1.

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{A}{\mapsto} x_1$	$x_4 \overset{B}{\mapsto} ch_5$
$x_0 \overset{I}{\mapsto} (x_1)\#(x_3)$	$x_{10} \overset{I}{\mapsto} (\backslash x_7.(\backslash x_8.x_7))$
$x_1 \overset{B}{\mapsto} ch_6$	$ch_6 \overset{A}{\mapsto} (x_{10})(x_4)$
$x_2 \overset{I}{\mapsto} (\backslash t.(\backslash t'.t))$	
$x_3 \overset{I}{\mapsto} 1$	
$ch_5 \overset{A}{\mapsto} x_3$	

La variable x_1 se encuentra ligada a un canal que aún no ha terminado su misión, por lo que la creación del segundo proceso está retenida por una comunicación pendiente. □

6.2.2.2. Comunicación

Los procesos no son entes aislados en el sistema, sino que se relacionan entre ellos enviándose valores a través de canales de comunicación. El modo de comunicarse queda formalizado en la Figura 6.5:

(value communication)
si $\mathsf{nf}(W, H_p) = \emptyset$
$freshrenaming(\eta)$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{com}{\longrightarrow} (S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathsf{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} \eta(W)\}\rangle)$$

Figura 6.5: Jauja básico: comunicación entre procesos

Condición indispensable para proceder a la comunicación de un valor, W, es que $\mathsf{nf}(W, H_p) = \emptyset$, es decir, que todas las ligaduras que haya que copiar tengan sus expresiones en *whnf*, y que no haya dependencia de canales ni de creaciones de proceso. Cuando todas las dependencias han sido resueltas, se copian todas las ligaduras tras ser

renombradas por (η). Estas condiciones son las que asumiremos cuando digamos que la *comunicación es factible*:

Definición 6.5 Sean S un sistema, H_1 y H_2 dos *heaps* de S, $ch \overset{\alpha}{\mapsto} W$ una ligadura de H_1 y $\theta \overset{B}{\mapsto} ch$ una ligadura de H_2. La comunicación a través del canal ch es una *comunicación factible* si $\mathsf{nf}(W, H_1) = \emptyset$.

□

La comunicación de un valor puede dar lugar a más comunicaciones. Esta regla global de paso simple tiene que ser aplicada repetidamente hasta que no sea posible ninguna comunicación. Como consecuencia, la regla de paso reiterado para comunicación se define:

$$\overset{com}{\Longrightarrow}$$

De nuevo, el orden de las aplicaciones de $\overset{com}{\longrightarrow}$ es irrelevante, pues la realización de una comunicación factible nunca impide la realización de otra que ya era factible antes de realizar la primera comunicación.

Notación 6.3 Una *ligadura-esperando-canal* es una ligadura de la forma $x \overset{B}{\mapsto} ch$.

□

Proposición 6.4 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} W_1 \in H_1$ una ligadura tal que la comunicación a través de ch_1 es factible. Sea H_2 otro heap cualquiera de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} W_2 \in H_2$ una ligadura tal que la comunicación a través de ch_2 es factible y con $ch_1 \neq ch_2$. Entonces la realización de la comunicación a través de ch_1 no impide la posterior realización de la comunicación a través de ch_2.*

Demostración P.6.4
La realización de la comunicación de ch_1 hace que esta ligadura desaparezca y que en el proceso consumidor la ligadura bloqueada en este canal quede activa una vez recibido el valor *whnf* comunicado, adecuadamente renombrado.

Si $H_1 = H_2$, antes de la comunicación por ch_1, tendríamos $\mathsf{nf}(W_2, H_2) = \emptyset$, y después H_1 solamente ha cambiado en lo que hace referencia a la ligadura de ch_1, con lo que $\mathsf{nf}(W_2, H_2) = \emptyset$ y, por lo tanto, la comunicación a través de ch_2 es factible.

En el caso de que ambos *heaps* sean distintos, la única posibilidad de que H_2 se vea afectado por la comunicación a través de ch_1 es que H_2 sea el *heap* del proceso consumidor. En caso contrario, H_2 no cambia y, en consecuencia, $\mathsf{nf}(W_2, H_2) = \emptyset$. Si se tratara de que es el consumidor, $\mathsf{nf}(W_2, H_2)$ era vacío, por lo que ahora el hecho de que haya una variable más ligada a un valor *whnf*, junto con la incorporación de ligaduras frescas, no cambian este conjunto. En consecuencia, la comunicación a través de ch_2 sigue siendo factible.

c.q.d.

Ahora bien, la realización de una comunicación factible puede llevar a que otra que antes no lo era ahora sí pueda realizarse.

Ejemplo 6.11 *Comunicación que habilita otra comunicación.*

Dado el siguiente sistema con dos canales:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{11} \overset{I}{\mapsto} (\backslash x_8.(\backslash x_9.x_8))$
$x_1 \overset{I}{\mapsto} (\backslash s.(\backslash t.s))$	$ch_7 \overset{I}{\mapsto} (\backslash x_9.x_5)$
$x_2 \overset{I}{\mapsto} 1$	
$x_3 \overset{I}{\mapsto} 2$	
$x_4 \overset{I}{\mapsto} (\backslash t'.(2))$	
$ch_6 \overset{A}{\mapsto} 2$	

la comunicación a través del canal ch_6 posibilita que la del canal ch_7 pueda realizarse.

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{A}{\mapsto} 2$
$x_0 \overset{A}{\mapsto} (\backslash x_{13}.x_{15})$	$x_{11} \overset{I}{\mapsto} (\backslash x_8.(\backslash x_9.x_8))$
$x_1 \overset{I}{\mapsto} (\backslash s.(\backslash t.s))$	
$x_2 \overset{I}{\mapsto} 1$	
$x_3 \overset{I}{\mapsto} 2$	
$x_4 \overset{I}{\mapsto} (\backslash t'.(2))$	
$x_{15} \overset{I}{\mapsto} 2$	

Obsérvese que al comunicar, a través de través de ch_7, en el *heap* del receptor se tiene que copiar la ligadura correspondiente a la variable libre contenida en la expresión ligada a x_{11}; esta variable es renombrada como x_{15} en el *heap* receptor. El lector también podrá percatarse de que los canales han desaparecido, pues ya han cumplido su misión de vías de comunicación y, como se formalizó en la regla VALUE-COMMUNICATION, ahora se eliminan de los correspondientes *heaps*.

□

Al igual que hicimos con $\overset{pc}{\Longrightarrow}$, veamos que la aplicación de $\overset{com}{\Longrightarrow}$ siempre termina cuando se parte de un sistema finito.

Proposición 6.5 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{com}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{com}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.5
En la Sección A.2.

Como siempre tenemos sistemas finitos partiendo de uno inicial finito, se tiene que $\overset{com}{\Longrightarrow}$ siempre termina.

En los siguientes ejemplos se muestra cómo proceden las comunicaciones.

Ejemplo 6.12 *Comunicaciones factibles.*

Consideremos un ejemplo en el que, tras la obtención de los valores asociados a los canales de comunicación, no es necesario esperar a la evaluación de variables libres para comunicar dichos valores.

$$\texttt{let } x = y\texttt{\#}z, y = \backslash t.t, z = \backslash t'.(1) \texttt{ in } x$$

main(N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = (y)\texttt{\#}(z), y = (\backslash t.t), z = (\backslash t'.(1)) \texttt{ in } x$

El primer paso es la reducción local de la expresión mediante LET:

main(N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} (x_1)\texttt{\#}(x_2)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$

Todas las dependencias están resueltas, por lo que se puede crear el nuevo proceso:

main(N° Hijos: 1)	$\mathbf{p_1}$(N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$	$x_3 \overset{B}{\mapsto} ch_4$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{A}{\mapsto} (x_8)(x_3)$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	
$ch_4 \overset{A}{\mapsto} x_2$	

La regla DEMAND bloquea la ligadura *main*.

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{B}{\mapsto} ch_4$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{A}{\mapsto} (x_8)(x_3)$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	
$ch_4 \overset{A}{\mapsto} x_2$	

Se hace necesaria la evaluación del canal ch_5 mediante una β-reducción.[3]

[3] En estos ejemplos aún no se están considerando las evaluaciones de ligaduras que no son demandadas por la ligadura *main* para su evaluación, pues las normas de planificación serán incluidas más adelante. Será entonces cuando se detallen las posibles ejecuciones en paralelo de distintas ligaduras.

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{B}{\mapsto} ch_4$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{A}{\mapsto} x_3$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	
$ch_4 \overset{A}{\mapsto} x_2$	

De nuevo se bloquea la ligadura en tratamiento porque demanda el valor que se asociará a la variable x_3:

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{B}{\mapsto} ch_4$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{B}{\mapsto} x_3$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	
$ch_4 \overset{A}{\mapsto} x_2$	

Mediante la transición local dirigida por VALUE el canal ch_4 queda ligado a un valor *whnf*:

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{B}{\mapsto} ch_4$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{B}{\mapsto} x_3$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	
$ch_4 \overset{A}{\mapsto} (\backslash t'.(1))$	

En este punto, el valor del canal ch_4 puede ser comunicado, pues no tiene dependencia de ninguna variable libre, , desapareciendo entonces el canal por el que se realiza la comunicación:

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{A}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{B}{\mapsto} x_3$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

Tras varias transiciones dirigidas por el planificador, que serán detalladas en el siguiente apartado, el sistema queda como sigue:

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{I}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{A}{\mapsto} x_3$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

La transición VALUE provoca que el canal ch_5 tenga asociado un valor *whnf*.

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{I}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{B}{\mapsto} ch_5$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_5 \overset{A}{\mapsto} (\backslash x_9.(1))$
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

Se procede entonces a la comunicación de dicho valor:

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{B}{\mapsto} x_0$	$x_3 \overset{I}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{A}{\mapsto} (\backslash x_{11}.(1))$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

De nuevo se obvian los pasos del planificador que nos conducen a

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{A}{\mapsto} x_0$	$x_3 \overset{I}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{I}{\mapsto} (\backslash x_{11}.(1))$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

El último paso local es el traspaso del valor ligado a x_0 a la variable *main*:

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{A}{\mapsto} (\backslash x_{11}.(1))$	$x_3 \overset{I}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{I}{\mapsto} (\backslash x_{11}.(1))$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

Tras los pasos del planificador, el sistema final es

main(N° Hijos : 1)	$\mathbf{p_1}$(N° Hijos : 0)
$main \overset{I}{\mapsto} (\backslash x_{11}.(1))$	$x_3 \overset{I}{\mapsto} (\backslash x_9.(1))$
$x_0 \overset{I}{\mapsto} (\backslash x_{11}.(1))$	$x_8 \overset{I}{\mapsto} (\backslash x_6.x_6)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	
$x_2 \overset{I}{\mapsto} (\backslash t'.(1))$	

□

Ejemplo 6.13 *Comunicación no factible.*

Consideremos una expresión en la que durante su evaluación se tiene un canal ligado a un valor *whnf*, pero que no puede ser aún comunicado por no tener resueltas sus

dependencias libres.

$$\texttt{let } x = y\texttt{\#}z, y = \backslash t.t, z = \backslash t'.s, s = y\, l, l = 1 \texttt{ in } x$$

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = (y)\texttt{\#}(z), y = (\backslash t.t), z = (\backslash t'.s), s = (y)(l), l = 1 \texttt{ in } x$

Como en todos los casos, el primer paso consiste en evaluar la declaración local para incorporar todas las variables, adecuadamente renombradas, en el *heap* del proceso principal:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} (x_1)\texttt{\#}(x_2)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$
$x_4 \overset{I}{\mapsto} 1$

Como no hay dependencia de x_1 con respecto a ninguna variable libre, se puede crear el nuevo proceso:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{A}{\mapsto} (x_{10})(x_5)$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$	
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{A}{\mapsto} x_2$	

La variable *main* se bloquea porque x_0 no tiene asociado un valor, i.e. se produce la demanda de esta evaluación:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{A}{\mapsto} (x_{10})(x_5)$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$	
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{A}{\mapsto} x_2$	

La demanda de la variable *main* se propaga al canal ch_7, y se lleva a cabo la β-reducción:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{A}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$	
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{A}{\mapsto} x_2$	

La evaluación del canal ch_7 provoca el bloqueo de su ligadura:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{B}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$	
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{A}{\mapsto} x_2$	

La demanda llega al canal ch_6. La regla VALUE le asocia un valor $whnf$:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{B}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$	
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{A}{\mapsto} (\backslash t'.x_3)$	

La comunicación a través de ch_6 no es posible porque el valor que habría que comunicar depende de x_3, que aún no está ligada a un valor $whnf$: $\mathsf{nf}(\backslash t'.x_3, \mathrm{main}) = \{x_3 \overset{I}{\mapsto} x_1\ x_4\}$. Es necesario demandar la evaluación de esta variable. Esta función la desarrollan las reglas del apartado siguiente, produciendo como resultado sistemas como el siguiente, en el que la ligadura en cuestión está activa:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{B}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} (\backslash t'.x_3)$	
$x_3 \overset{A}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{I}{\mapsto} (\backslash t'.x_3)$	

Tras la evaluación de la ligadura de x_3 se continuaría con la comunicación pertinente. □

6.2.2.3. Planificación

Cuando ya se hayan realizado todas las transiciones locales que corresponda, así como las creaciones de proceso y comunicaciones que sean posibles, se tienen que reorganizar las etiquetas de las ligaduras del sistema. Las tareas que hay que llevar a cabo son:

- Desbloqueo de ligaduras dependientes de una variable que ya tiene asociado un valor.
- Desactivación de las ligaduras que ya han alcanzado una *whnf*.
- Bloqueo de las creaciones de proceso que no han podido realizarse.
- Demanda de la evaluación de las ligaduras necesarias para llevar a cabo:
  - Creaciones de proceso pendientes
  - Comunicaciones no realizadas.

Todas estas operaciones quedan recogidas en las reglas de la Figura 6.6, en la que en particular se utiliza la Notación 6.2.

(WHNF unblocking)

$$(S, \langle p, H + \{x \overset{A}{\mapsto} W, \theta \overset{B}{\mapsto} E_B^x\}\rangle) \overset{wUnbl}{\longrightarrow} (S, \langle p, H + \{x \overset{A}{\mapsto} W, \theta \overset{A}{\mapsto} E_B^x\}\rangle)$$

(WHNF deactivation)

$$(S, \langle p, H + \{\theta \overset{A}{\mapsto} W\}\rangle) \overset{deact}{\longrightarrow} (S, \langle p, H + \{\theta \overset{I}{\mapsto} W\}\rangle)$$

(blocking process creation)

$$(S, \langle p, H + \{\theta \overset{IA}{\mapsto} x\#y\}\rangle) \overset{bpc}{\longrightarrow} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x\#y\}\rangle)$$

(process creation demand)

si $y \overset{I}{\mapsto} E \in \mathtt{nf}(x, H)$

$$(S, \langle p, H + \{\theta \overset{B}{\mapsto} x_1\#x_2\}\rangle) \overset{pcd}{\longrightarrow} (S, \langle p, H + \{\theta \overset{B}{\mapsto} x_1\#x_2, y \overset{A}{\mapsto} E\}\rangle)$$

(value communication demand)

si $x \overset{I}{\mapsto} E \in \mathtt{nf}(W, H)$

$$(S, \langle p, H + \{ch \overset{I}{\mapsto} W\}\rangle) \overset{vComd}{\longrightarrow} (S, \langle p, H + \{ch \overset{I}{\mapsto} W, x \overset{A}{\mapsto} E\}\rangle)$$

Figura 6.6: Jauja básico: planificación

El desbloqueo de una ligadura tiene lugar cuando la variable de la que depende ya está ligada a un valor *whnf*. Como ya se puede proseguir con su evaluación, la regla WHNF-UNBLOCKING cambia su etiqueta de bloqueada a activa. Obsérvese que en el conjunto E_B^x no pueden aparecer ni ch ni $ch\ y$. Ocurrencias del segundo tipo no pueden

aparecer por la normalización de las expresiones y porque un canal solamente aparece en ligaduras de la forma $\theta \overset{\alpha}{\mapsto} ch$ o bien $ch \overset{\alpha}{\mapsto} E$, con $E \notin \mathcal{C}han$. La primera posibilidad tampoco es viable porque cuando una ligadura está bloqueada en un canal la ligadura de este canal no aparece en el mismo *heap* sino que está en otro proceso.

Varias ligaduras pueden estar pendientes de la obtención de un mismo valor *whnf*. Cuando se ha producido dicho valor todas ellas son desbloqueadas por la regla anterior. Tras este desbloqueo, la ligadura del valor *whnf* se desactiva, pudiéndose posteriormente aplicar las reglas locales DEMAND y APP-DEMAND. Gracias a la regla WHNF DEACTIVATION se desactivan —es decir, las etiquetas cambian de activas a inactivas— todas las ligaduras que acaban de ligarse a valor *whnf*.

Ejemplo 6.14 *Desbloqueo y desactivación de ligaduras.*

En la siguiente expresión se producen los dos tipos de bloqueo: una variable demanda directamente otra, y una aplicación funcional necesita de la evaluación de la variable de la abstracción.

$$\texttt{let } x = (y)(z), y = (s)(t), z = 1, s = (\backslash u.u), t = (\backslash u.u) \texttt{ in } x$$

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = (y)(z), y = (s)(t), z = 1, s = (\backslash u.u), t = (\backslash u.u) \texttt{ in } x$

LET

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} (x_1)(x_2)$
$x_1 \overset{I}{\mapsto} (x_3)(x_4)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

DEMAND sobre $main$ y x_0

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{A}{\mapsto} (x_1)(x_2)$
$x_1 \overset{I}{\mapsto} (x_3)(x_4)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

APP-DEMAND sobre x_0 y x_1

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{B}{\mapsto} (x_1)(x_2)$
$x_1 \overset{A}{\mapsto} (x_3)(x_4)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

Se tiene una ligadura bloqueada por DEMAND ligada directamente a una variable, y la regla APP-DEMAND ha bloqueado la segunda ligadura.

β-REDUCCIÓN

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{B}{\mapsto} (x_1)(x_2)$
$x_1 \overset{A}{\mapsto} x_4$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

VALUE

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{B}{\mapsto} (x_1)(x_2)$
$x_1 \overset{A}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

WHNF UNBLOCKING originado por x_1

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{A}{\mapsto} (x_1)(x_2)$
$x_1 \overset{A}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

Ya se ha desbloqueado una de las ligaduras. Ahora se desactivan todas las ligaduras activas que ya están ligadas a valor *whnf*, para poder ejecutar posteriormente la regla local β-REDUCTION.

WHNF DEACTIVATION

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{A}{\mapsto} (x_1)(x_2)$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

β-REDUCTION

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{A}{\mapsto} x_2$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

VALUE

main (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{A}{\mapsto} 1$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

WHNF UNBLOCKING originado por x_0

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{A}{\mapsto} 1$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

WHNF DEACTIVATION

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} 1$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

VALUE

main (N° Hijos: 0)
$main \overset{A}{\mapsto} 1$
$x_0 \overset{I}{\mapsto} 1$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

Para finalizar la evaluación solamente resta desactivar la variable *main*:

main (N° Hijos: 0)
$main \overset{I}{\mapsto} 1$
$x_0 \overset{I}{\mapsto} 1$
$x_1 \overset{I}{\mapsto} (\backslash u.u)$
$x_2 \overset{I}{\mapsto} 1$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$
$x_4 \overset{I}{\mapsto} (\backslash u.u)$

□

La causa de que una creación de proceso no sea realizada durante la aplicación de $\overset{pc}{\Longrightarrow}$, la causa es la dependencia de variables libres asociadas a ligaduras que aún no han alcanzado una *whnf*. En consecuencia, la ligadura de la **#**-expresión se bloquea con la regla BLOCKING PROCESS CREATION y mediante la regla PROCESS CREATION DEMAND se demandan todas las ligaduras de las que depende que estén inactivas y aún no en *whnf*.

Ejemplo 6.15 *Creación de proceso no factible: demanda de evaluación.*

Retomemos la evaluación del Ejemplo 6.6, para bloquear la ligadura de la creación de proceso y demandar la evaluación de la variable correspondiente. En el último sistema de aquel ejemplo teníamos:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{I}{\mapsto} (x_1)\#(x_2)$
$x_1 \overset{I}{\mapsto} (\backslash s.(x_2))$
$x_2 \overset{I}{\mapsto} (x_3)(x_3)$
$x_3 \overset{I}{\mapsto} (\backslash u.u)$

y se bloquea la creación de proceso. El siguiente paso de la reducción es la demanda de evaluación de la variable x_1:

BLOCKING PROCESS CREATION

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{B}{\mapsto} (x_1)\#(x_2)$
$x_1 \overset{I}{\mapsto} (x_2)(x_2)$
$x_2 \overset{I}{\mapsto} (\backslash s.s)$

PROCESS CREATION DEMAND

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_0$
$x_0 \overset{B}{\mapsto} (x_1)\#(x_2)$
$x_1 \overset{A}{\mapsto} (x_2)(x_2)$
$x_2 \overset{I}{\mapsto} (\backslash s.s)$

□

De manera similar, si una comunicación no ha podido realizarse es a causa de que algunas de sus dependencias no están preparadas aún. La regla VALUE COMMUNICATION DEMAND activa todas las ligaduras involucradas que aún están inactivas y no han alcanzado la forma *whnf*. Nótese que la ligadura del canal ch a W se encuentra inactiva por el proceso anterior de desactivación.

Ejemplo 6.16 *Comunicación no factible: demanda de evaluación.*

Recordemos el último paso del Ejemplo 6.13 y veamos qué paso de demanda se realiza:

main (N° Hijos: 1)
$main \overset{B}{\mapsto} x_0$
$x_0 \overset{B}{\mapsto} ch_7$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$
$x_2 \overset{I}{\mapsto} (\backslash u.x_3)$
$x_3 \overset{I}{\mapsto} (x_1)(x_4)$
$x_4 \overset{I}{\mapsto} 1$
$ch_6 \overset{A}{\mapsto} (\backslash u.x_3)$

p_1 (N° Hijos: 0)
$x_5 \overset{B}{\mapsto} ch_6$
$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$ch_7 \overset{B}{\mapsto} x_5$

La comunicación a través de ch_6 no era posible porque el valor que habría que comunicar depende de x_3, aún sin ligar a un valor *whnf*. Aplicando las reglas WHNF DEACTIVATION y VALUE COMMUNICATION DEMAND se obtiene el sistema:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{B}{\mapsto} ch_6$
$x_0 \overset{B}{\mapsto} ch_7$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{I}{\mapsto} (\backslash t.t)$	$ch_7 \overset{B}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} (\backslash u.x_3)$	
$x_3 \overset{A}{\mapsto} (x_1)(x_4)$	
$x_4 \overset{I}{\mapsto} 1$	
$ch_6 \overset{I}{\mapsto} (\backslash u.x_3)$	

Se ha desactivado la ligadura del canal de comunicación, que ya ha alcanzado la forma *whnf*, pero no ha podido comunicarse. Por otro lado, se ha activado la ligadura de la variable x_3, pues hasta que esta variable no se encuentre ligada a un valor *whnf* no se podrá comunicar a través de ch_6.

□

La iteración de las reglas de la Figura 6.6 da lugar a una nueva regla global:

$$\overset{Unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow} .$$

Para cada una de las reglas de paso reiterado que componen $\overset{Unbl}{\Longrightarrow}$, se demuestra en la Sección A.3 que un paso simple no inhabilita otra aplicación que era posible antes, con lo que el orden de aplicación de cada uno de los pasos simples que componen cada una no afecta al resultado final del paso reiterado.

Pasemos a demostrar que $\overset{Unbl}{\Longrightarrow}$ está bien definida. Para ello basta con demostrar que cada una de las transiciones globales que la componen también está bien definida. Las proposiciones correspondientes están incluidas en la Sección A.4. De todas ellas se deduce la siguiente proposición:

Proposición 6.6 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{Unbl}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{Unbl}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.6
Por las Proposiciones A.13, A.14, A.15, A.16 y A.17 cada paso reiterado que compone $\overset{Unbl}{\Longrightarrow}$ está bien definido. Si todas las imágenes para cualquier sistema finito están definidas, la aplicación sucesiva de los cinco pasos reiterados a partir de S darán lugar a una imagen bien definida, es decir, la imagen para S por $\overset{Unbl}{\Longrightarrow}$ está bien definida.

c.q.d.

Por el momento, todas las transiciones compuestas por la aplicación reiterada de transiciones globales están bien definidas. Falta ver cómo evoluciona el sistema en conjunto y ver que esta evolución también se realiza de manera adecuada.

6.2.2.4. Evolución del sistema

Después de que cada proceso evolucione de manera local, o interna a él mismo, el sistema global también evoluciona. Esta progresión de la evaluación queda recogida por la aplicación de cada una de las transiciones globales que hemos visto anteriormente, quedando la regla global de evolución del sistema definida en la forma siguiente:

$$\overset{sys}{\Longrightarrow} = \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{Unbl}{\Longrightarrow}$$

En primer lugar se llevan a cabo todas las comunicaciones posibles. El segundo paso global desarrolla todas las creaciones de proceso que sean factibles. Este orden no es aleatorio, pues la realización de una comunicación puede liberar una creación de proceso pendiente, en tanto que la simple creación de un proceso no va a liberar ninguna comunicación pendiente. Además, cuando se crea un nuevo proceso no es posible realizar ninguna comunicación si antes no se ha realizado como mínimo la transición local VALUE: debido a la normalización de las expresiones un canal queda inicialmente ligado a una variable, por ende no comunicable. El paso global del sistema finaliza con las tareas de desbloqueo, desactivación, bloqueo y demanda. Es natural que este paso sea el último, pues si no se ha evaluado nada, ningún cambio ha sido realizado y, por lo tanto, ninguna de estas acciones tiene sentido.

Ejemplo 6.17 *Comunicación que habilita una creación de proceso.*

En el siguiente sistema se tiene bloqueada la creación de proceso de x_0, pues se espera que se comunique un valor a través de ch_7:

main(N°. Hijos: 1)	$\mathbf{p_1}$(N°. Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{I}{\mapsto} 1$
$x_0 \overset{B}{\mapsto} (x_2)\#(x_3)$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{B}{\mapsto} ch_7$	$ch_7 \overset{A}{\mapsto} 1$
$x_2 \overset{I}{\mapsto} (\backslash s.x_1)$	
$x_3 \overset{I}{\mapsto} 1$	
$x_4 \overset{I}{\mapsto} (\backslash t'.t')$	

La comunicación a través de ch_7 es posible, y se procede a ella:

main(N°. Hijos: 1)	$\mathbf{p_1}$(N°. Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{I}{\mapsto} 1$
$x_0 \overset{B}{\mapsto} (x_2)\#(x_3)$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$
$x_1 \overset{A}{\mapsto} 1$	
$x_2 \overset{I}{\mapsto} (\backslash s.x_1)$	
$x_3 \overset{I}{\mapsto} 1$	
$x_4 \overset{I}{\mapsto} (\backslash t'.t')$	

Ahora sí que es posible la creación del proceso:

main(N°. Hijos: 2)	$\mathbf{p_1}$(N°. Hijos: 0)	$\mathbf{p_2}$(N°. Hijos: 0)
$main \overset{B}{\mapsto} x_0$	$x_5 \overset{I}{\mapsto} 1$	$x_{13} \overset{B}{\mapsto} ch_{14}$
$x_0 \overset{B}{\mapsto} ch_{15}$	$x_{10} \overset{I}{\mapsto} (\backslash x_8.x_8)$	$x_{18} \overset{I}{\mapsto} (\backslash x_{16}.x_{19})$
$x_1 \overset{A}{\mapsto} 1$		$x_{19} \overset{I}{\mapsto} 1$
$x_2 \overset{I}{\mapsto} (\backslash s.x_1)$		$ch_{15} \overset{A}{\mapsto} (x_{18})(x_{13})$
$x_3 \overset{I}{\mapsto} 1$		
$x_4 \overset{I}{\mapsto} (\backslash t'.t')$		
$ch_{14} \overset{A}{\mapsto} x_3$		

□

Finalmente, cada paso de transición del sistema se define mediante:

$$\Longrightarrow \; = \overset{par}{\Longrightarrow} ; \overset{sys}{\Longrightarrow}$$

donde $\overset{par}{\Longrightarrow}$, que recoge la evolución local de cada proceso, se define a continuación.

6.2.3. Evolución paralela considerando especulación

Hasta ahora hemos creado procesos, pero la evaluación que hemos observado en todos los ejemplos era secuencial. El paralelismo no sólo consiste en la creación de procesos paralelos, sino que también se preocupa de la ejecución en paralelo de distintas ligaduras activas. Sin embargo, no siempre se explota todo el paralelismo introducido en el programa, pues estamos limitados por el número de procesadores disponibles, las decisiones del planificador y la velocidad de las instrucciones básicas. En cualquier caso, el paralelismo se mueve en un rango que va desde la mínima expresión del mismo: solamente se evalúa lo que es necesario para obtener un valor *whnf* a partir de la expresión (programa) inicial; a la explotación de la totalidad del paralelismo del programa, desarrollando todo el paralelismo especulativo introducido. Es por eso que aquí se van a considerar estos dos casos extremos, como muestras representativas del amplio rango posible de modos de evaluación. Para proceder a ello, definamos previamente el concepto de secuencia de reducción:

Definición 6.6 Dada una expresión-programa E, se define una *secuencia de reducción* para dicha expresión como una secuencia, finita o infinita, de sistemas tales que:

$$\langle p_0, \{main \overset{A}{\mapsto} E\}\rangle \Longrightarrow^* \langle p_0, H + \{main \overset{I}{\mapsto} W\}\rangle, \langle p_1, H_1\rangle, \ldots, \langle p_n, H_n\rangle \Longrightarrow^*$$

□

La configuración (sistema) final depende de la semántica en consideración:

Mínima: la primera en la que la ligadura de la variable *main* esté inactiva.

Máxima: no existe ninguna ligadura activa, es decir, todos los procesos creados han terminado su ejecución, o el sistema está bloqueado sin posibilidad alguna de evolución.

Cada proceso evoluciona en su seno interno llevando a cabo las transiciones locales de la Figura 6.2, de manera que el progreso local de un proceso p queda determinado por el conjunto de ligaduras paralelas activas en su *heap* H_p a las que se permite evolucionar.

Definición 6.7 Dado un *heap* H, se define el *conjunto de ligaduras desarrollables de* H, $\mathcal{LD}(H)$, como el formado por las ligaduras activas de H a las que se permite evolucionar aplicando una regla local (Figura 6.2).

□

La evolución del proceso p se consigue aplicando a cada ligadura de dicho conjunto una regla local:

$$\textbf{(parallel-p)}$$

$$\frac{\{H_p^{(i,1)} + H_p^{(i,2)} : \theta_p^i \overset{A}{\mapsto} E_p^i \longrightarrow H_p^{(i,1)} + K_p^{(i,2)} | H_p = H_p^{(i,1)} + H_p^{(i,2)} + \{\theta_p^i \overset{A}{\mapsto} E_p^i\} \wedge \theta_p^i \overset{A}{\mapsto} E_p^i \in \mathcal{LD}(H_p)\}_{i=1}^{n_p}}{\langle p, H_p\rangle \overset{par}{\Longrightarrow}_p \langle p, (\cap_{i=1}^{n_p} H_p^{(i,1)}) \cup (\cup_{i=1}^{n_p} K_p^{(i,2)})\rangle}$$

donde $n_p = |\mathcal{LD}(H_p)|$, $H_p^{(i,1)}$ es la parte de H_p que permanece sin modificar tras la aplicación de la correspondiente regla local, y $K_p^{(i,2)}$ contiene las ligaduras modificadas de H_p (más concretamente, de $H_p^{(i,2)}$).

Ejemplo 6.18 *Evolución en paralelo de ligaduras de un mismo proceso.*

Supongamos que las ligaduras desarrollables del siguiente *heap* son todas las activas existentes:

p (N° Hijos: 0)
- $main \overset{A}{\mapsto} x_7$
- $x_0 \overset{B}{\mapsto} (x_3)\#(x_5)$
- $x_1 \overset{B}{\mapsto} (x_4)\#(x_5)$
- $x_2 \overset{I}{\mapsto} (\backslash t.t)$
- $x_3 \overset{A}{\mapsto} (x_2)(x_2)$
- $x_4 \overset{A}{\mapsto} (x_2)(x_5)$
- $x_5 \overset{I}{\mapsto} 1$
- $x_6 \overset{I}{\mapsto} (\backslash t.(2))$
- $x_7 \overset{I}{\mapsto} (x_2)(x_6)$

En este ejemplo $\mathcal{LD}(H_p) = \{main, x_3, x_4\}$. Estas variables evolucionan en paralelo: *main*, aplicándole la regla DEMAND, y x_3 y x_4, aplicando a ambas la regla β-REDUCTION. Las componentes de la regla PARALLEL-P para la primera variable de $\mathcal{LD}(H_p)$, *main*, son:

$\mathbf{H_p^{(1,1)}}$	$\mathbf{H_p^{(1,2)}}$	$\mathbf{K_p^{(1,2)}}$
$x_0 \overset{B}{\mapsto} (x_3)\#(x_5)$	$main \overset{A}{\mapsto} x_7$	$main \overset{B}{\mapsto} x_7$
$x_1 \overset{B}{\mapsto} (x_4)\#(x_5)$	$x_7 \overset{I}{\mapsto} (x_2)(x_6)$	$x_7 \overset{A}{\mapsto} (x_2)(x_6)$
$x_2 \overset{I}{\mapsto} (\backslash t.t)$		
$x_3 \overset{A}{\mapsto} (x_2)(x_2)$		
$x_4 \overset{A}{\mapsto} (x_2)(x_5)$		
$x_5 \overset{I}{\mapsto} 1$		
$x_6 \overset{I}{\mapsto} (\backslash t.(2))$		

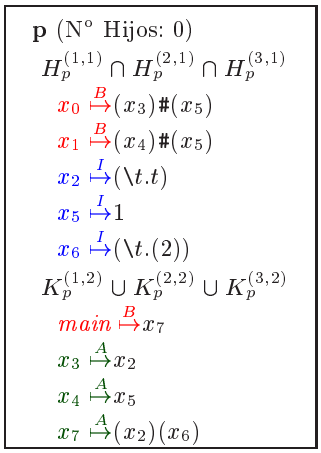
Para la segunda variable de $\mathcal{LD}(H_p)$, x_3:

$\mathbf{H_p^{(2,1)}}$	$\mathbf{H_p^{(2,2)}}$	$\mathbf{K_p^{(2,2)}}$
$main \overset{A}{\mapsto} x_7$	$x_3 \overset{A}{\mapsto} (x_2)(x_2)$	$x_3 \overset{A}{\mapsto} (x_2)$
$x_0 \overset{B}{\mapsto} (x_3)\#(x_5)$		
$x_1 \overset{B}{\mapsto} (x_4)\#(x_5)$		
$x_2 \overset{I}{\mapsto} (\backslash t.t)$		
$x_4 \overset{A}{\mapsto} (x_2)(x_5)$		
$x_5 \overset{I}{\mapsto} 1$		
$x_6 \overset{I}{\mapsto} (\backslash t.(2))$		
$x_7 \overset{I}{\mapsto} (x_2)(x_6)$		

Y para la tercera variable de $\mathcal{LD}(H_p)$, x_4:

$\mathbf{H_p^{(3,1)}}$	$\mathbf{H_p^{(3,2)}}$	$\mathbf{K_p^{(3,2)}}$
$main \overset{A}{\mapsto} x_7$	$x_4 \overset{A}{\mapsto} (x_2)(x_5)$	$x_4 \overset{A}{\mapsto} (x_5)$
$x_0 \overset{B}{\mapsto} (x_3)\#(x_5)$		
$x_1 \overset{B}{\mapsto} (x_4)\#(x_5)$		
$x_2 \overset{I}{\mapsto} (\backslash t.t)$		
$x_3 \overset{A}{\mapsto} (x_2)(x_5)$		
$x_5 \overset{I}{\mapsto} 1$		
$x_6 \overset{I}{\mapsto} (\backslash t.(2))$		
$x_7 \overset{I}{\mapsto} (x_2)(x_6)$		

Siendo el *heap* final:



□

En este ejemplo se observa que no hay interferencia entre cada una de las evoluciones locales, es decir, las ligaduras que modifica una evolución no se ven involucradas en la evolución de las otras ligaduras. La corrección de la regla PARALLEL-P pasa por ver que cada ligadura activa puede evolucionar en paralelo sin interferencia con ninguna otra ligadura activa del mismo proceso:

Para demostrar la Proposición 6.7, que asegura la no interferencia entre evoluciones locales, se necesitan dos lemas previos. Estos lemas establecen que la única ligadura activa modificada por una regla local es la que guía la aplicación de dicha regla (Lema 6.1), y que para cada ligadura activa existe a lo sumo una regla local aplicable sobre ella (Lema 6.2).

Lema 6.1 *Si* $H^1 + H^2 : \theta \overset{A}{\mapsto} E \longrightarrow H^1 + K$ *y* $\theta' \overset{A}{\mapsto} E' \in H^1 \cup H^2$ *con* $\theta \neq \theta'$*, entonces* $\theta' \overset{A}{\mapsto} E' \notin K$*.*

Demostración L. 6.1
Si observamos las reglas locales de la Figura 6.2, cuando evoluciona una ligadura activa, solamente si se aplican DEMAND o APP-DEMAND se encuentra involucrada otra ligadura activa del *heap*. Sin embargo, la aplicación de estas reglas únicamente modifica a dicha ligadura activa.

c.q.d.

Lema 6.2 *Sea* $\theta \overset{A}{\mapsto} E \in H$ *una ligadura activa. Entonces existe a lo sumo una regla local aplicable a ella.*

Demostración L. 6.2
En primer lugar, la observación "a lo sumo" viene motivada por la existencia de #-expresiones —que no evolucionan por medio de ninguna regla local— así como sucede

con las abstracciones, que por ser valores *whnf* no tienen que evolucionar de manera local, y a lo sumo se procede a su desactivación en pasos globales. Para el resto de las expresiones existe una única regla local aplicable a cada una de ellas:

E es una variable: caben tres posibilidades, que la ligadura de la variable sea la misma que la activa a evolucionar, que sea distinta y esté inactiva en un valor *whnf* o que sea distinta y esté ligada a una expresión no en *whnf*. En el primer caso solamente se puede aplicar la regla BLACKHOLE, en el segundo la única regla local aplicable es VALUE y en el tercero la única posibilidad es aplicar la regla DEMAND.

E es una aplicación funcional: se puede estar en dos situaciones, que aún no se haya obtenido una abstracción para la primera variable o que dicha variable sí se encuentre ligada a una abstracción. En el primer caso solamente se puede aplicar la regla APP-DEMAND para pedir que se evalúe esta variable. En el segundo caso sólo cabe la posibilidad de proceder con la β-REDUCTION.

E es una declaración local de variables: la única regla local que se puede aplicar es LET.

En conclusión, a lo sumo se puede aplicar una regla local a la ligadura $\theta \overset{A}{\mapsto} E$.

c.q.d.

Proposición 6.7 *Consideremos dos variables distintas, θ_1 y θ_2, y un heap que puede expresarse de dos modos distintos: $H = H^{(i,1)} + H^{(i,2)} + \{\theta_i \overset{A}{\mapsto} E_i\}$, con $i \in \{1,2\}$, de modo que en cada caso tengamos $H^{(i,1)} + H^{(i,2)} : \theta_i \overset{A}{\mapsto} E_i \longrightarrow H^{(i,1)} + K^{(i,2)}$. Entonces*

1. *Si $\theta' \overset{IB}{\mapsto} E' \in K^{(1,2)}$ tendremos $\theta' \notin dom(K^{(2,2)})$*
2. *Si $\theta' \overset{A}{\mapsto} E' \in K^{(1,2)}$ tendremos $\theta' \notin dom(K^{(2,2)})$ o bien $\theta' \overset{A}{\mapsto} E' \in K^{(2,2)}$*

Demostración P.6.7
Distinguimos primero casos sobre la regla aplicada para $i = 1$:

- VALUE: entonces $K^{(1,2)} = \{\theta_1 \overset{A}{\mapsto} W\}$. Así, ninguna ligadura inactiva o bloqueada pertenece a este conjunto. Además, como $\theta_1 \neq \theta_2$, por el Lema 6.1 la ligadura de θ_1 no puede ser modificada por la evolución de θ_2.

- DEMAND: en este caso, $K^{(1,2)} \subseteq \{\theta_1 \overset{B}{\mapsto} x, x \overset{A}{\mapsto} E'\}$. Veamos qué puede suceder con x por inducción estructural sobre E_2 (considerando que la etiqueta anterior de la ligadura de x era inactiva, se deduce además que $x \neq \theta_2$):
  - En el caso de que E_2 sea una abstracción no existe una regla local aplicable.
  - Si $E_2 = y$, entonces solamente son aplicables tres reglas. Si es VALUE, entonces $K^{(2,2)} = \{\theta_2 \overset{A}{\mapsto} W\}$; y como $x \neq \theta_2$, $x \notin dom(K^{(2,2)})$. Si la regla es DEMAND, entonces $dom(K^{(2,2)}) = \{y, \theta_2\}$; la hipótesis inicial asegura que $x \neq \theta_2$, y si $x = y$ en todo caso $x \overset{A}{\mapsto} E' \in K^{(2,2)}$. Finalmente, si la regla es BLACKHOLE, entonces $K^{(2,2)} = \{\theta_2 \overset{B}{\mapsto} \theta_2\}$ y por hipótesis $x \neq \theta_2$, con lo que $x \notin dom(K^{(2,2)})$.

- Si $E_2 = \texttt{let}\ \{x^i = E^i\}_{i=1}^n\ \texttt{in}\ E'$, entonces aplicando la regla LET se tiene que $dom(K^{(2,2)}) = \{y^i\}_{i=1}^n \cup \{\theta_2\}$. $x \neq \theta_2$ y todas las variables y^i son frescas, de manera que $x \notin dom(K^{(2,2)})$.
- Si $E_2 = z\ y$ entonces
  - Si $z \overset{I}{\mapsto} \backslash x.E'$: la regla aplicable es β-REDUCTION, siendo entonces $K^{(2,2)} = \{\theta_2 \overset{A}{\mapsto} E'[y/x]\}$. Como $x \neq \theta_2$, entonces $x \notin dom(K^{(2,2)})$.
  - Si $z \overset{IAB}{\mapsto} = E_2^1$, y E_2^1 no es una abstracción. Entonces, la regla aplicable es APP-DEMAND, y $K^{(2,2)} \subseteq \{z \overset{A}{\mapsto} E_2^1, \theta_2 \overset{B}{\mapsto} z\ y\}$. Por hipótesis sabemos que $x \neq \theta_2$. Si $x \neq z$ entonces ya tenemos el resultado, pero si $x = z$, en ambos casos se tiene su ligadura modificada activada, con lo que también se llega al resultado deseado.

- BLACKHOLE: entonces $K^{(1,2)} = \{\theta_1 \overset{B}{\mapsto} \theta_1\}$, y por el Lema 6.1 la propiedad se cumple.
- LET: entonces $K^{(1,2)} = \{y^i \overset{I}{\mapsto} E^i\}_{i=1}^n \cup \{\theta_1 \overset{A}{\mapsto} E'\}$. Todas las variables x^i son nuevas, de manera que no pueden pertenecer al conjunto $K^{(2,2)}$, y el Lema 6.1 garantiza que la ligadura de θ_1 no puede ser modificada por la evolución de θ_2.
- β-REDUCTION: entonces $K^{(1,2)} = \{\theta_1 \overset{A}{\mapsto} E'[y/x]\}$. Así $\nexists \theta_1' \overset{IB}{\mapsto} E' \in K^{(1,2)}$, y el Lema 6.1 asegura que la propiedad se verifica.
- APP-DEMAND: en este caso $E_1 = x\ y$, siendo $K^{(1,2)} \subseteq \{x \overset{A}{\mapsto} E', \theta_1 \overset{B}{\mapsto} x\ y\}$. Igual que sucedía en el caso de la regla DEMAND, sólo tenemos que preocuparnos de lo que pueda suceder con x, y en particular, el caso en que x estuviera inactiva y ahora se hubiera activado, de donde se deduce que $x \neq \theta_2$. Un razonamiento análogo al aplicado para la regla DEMAND conduce al resultado.

Analizados todos los casos, se garantiza la no interferencia entre la evolución de distintas ligaduras activas.

c.q.d.

Vista la corrección de la regla PARALLEL-P, la evolución en paralelo de los procesos en un sistema S queda definida por la regla global:

(parallel)

$$\frac{\{\langle p, H_p\rangle \overset{par}{\Longrightarrow}_p \langle p, H_p'\rangle\}_{\langle p,H_p\rangle \in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H_p'\rangle\}_{\langle p,H_p\rangle \in S}}$$

Una vez definido cómo pueden evolucionar en paralelo los distintos procesos del sistema y las ligaduras de cada proceso (lo que da lugar a dos niveles distintos de paralelismo), vamos a estudiar las dos cotas que definen los límites de la cantidad de paralelismo explotado en el sistema.

Semántica mínima

Como se mencionó antes, en este caso no se desarrolla ninguna evaluación especulativa, solamente son evaluadas aquellas ligaduras que son demandadas. Esto conduce a introducir un mecanismo que controle que la evaluación no se pierda en tareas especulativas y dé preferencia al trabajo demandado por la variable principal (*main*) del proceso principal, i.e. el inicial. Esta tarea es realizada por la función pre (preferencia):

$$\mathsf{pre}(x,\langle p,H\rangle,S)=\begin{cases}\{\langle x,\langle p,H\rangle\rangle\}, & \text{si } x \overset{A}{\mapsto} E \in H;\\ \mathsf{pre}(y,\langle p,H\rangle,S), & \text{si } x \overset{B}{\mapsto} E^y_B \in H;\\ \bigcup\limits_{y\in\mathsf{nf}(E_1,H)} \mathsf{pre}(y,\langle p,H\rangle,S), & \text{si } x \overset{B}{\mapsto} E_1\#E_2 \in H;\\ \mathsf{pre}(ch,\langle p,H_{ch}\rangle,S), & \text{si } x \overset{B}{\mapsto} ch \in H \wedge ch \in dom(H_{ch}) \wedge H_{ch} \in S;\\ \emptyset, & \text{e.o.c.}\end{cases}$$

$$\mathsf{pre}(ch,\langle p,H\rangle,S)=\begin{cases}\{\langle ch,\langle p,H\rangle\rangle\}, & \text{si } ch \overset{A}{\mapsto} E \in H;\\ \mathsf{pre}(y,\langle p,H\rangle,S), & \text{si } ch \overset{B}{\mapsto} E^y_B \in H;\\ \bigcup\limits_{y\in\mathsf{nf}(E_1,H)} \mathsf{pre}(y,\langle p,H\rangle,S), & \text{si } ch \overset{B}{\mapsto} E_1\#E_2 \in H;\\ \mathsf{pre}(ch',\langle p,H_{ch'}\rangle,S), & \text{si } ch \overset{B}{\mapsto} ch' \in H \wedge ch' \in dom(H_{ch'}) \wedge H_{ch'} \in S;\\ \bigcup\limits_{y\in\mathsf{nf}(W,H)} \mathsf{pre}(y,\langle p,H\rangle,S), & \text{si } ch \overset{I}{\mapsto} W \in H;\\ \emptyset, & \text{e.o.c.}\end{cases}$$

Esta función devuelve el conjunto de ligaduras que demandan una variable ordinaria, x, o un canal, ch, según el caso, para su evaluación inmediata. Veamos los cuatro primeros casos de esta función, que son comunes para ambos:

- En el primer caso, la variable está ligada a una expresión y se encuentra activa, por lo que hay que proceder a evaluar esta misma ligadura.
- En los restantes casos la variable está bloqueada. Como ya se vio anteriormente al definir la función nf (y, ahora tras exponer la totalidad de reglas del sistema, resulta más evidente), esta situación se puede dar solamente por las siguientes razones:
  1. Se bloqueó por la aplicación de DEMAND o de APP-DEMAND.
  2. Se encuentra ligada a una creación de proceso que no pudo realizarse y se bloqueó con BLOCKING PROCESS CREATION.
  3. La variable está bloqueada en un canal, producto de las modificaciones del sistema derivadas de una creación de proceso.

  La primera posible causa se corresponde con el segundo caso de las definiciones de pre. Allí se busca, a partir de la variable y por la que está bloqueada nuestra variable, las ligaduras que pueden evolucionar. Para la segunda posible causa es preciso que se evalúen las dependencias libres de la abstracción de la creación, para así proceder con ésta. Finalmente, en el último supuesto la búsqueda se propaga al *heap* del productor del canal que bloquea a nuestra variable.

El caso que diferencia ambas definiciones de pre es aquél en el que canal ch está ligado a un valor *whnf*. Esto significa que la comunicación no se ha podido llevar a cabo, a falta de evaluar dependencias libres, y son estas dependencias las que tiene que recoger la función pre.

Para conseguir la evaluación mínima, la función anterior se particulariza para la variable principal:

Definición 6.8 Sea S un sistema en el que el proceso $\langle p_0, H_0\rangle$ es el único cuyo *heap* contiene la variable *main* ($\langle p_0, H_0\rangle$ es el proceso inicial). Se define la *demanda, o precedencia, de main* como $\mathsf{pm}(S) = \mathsf{pre}(main, \langle p_0, H_0\rangle, S)$.

□

El conjunto $\mathcal{LD}_{min} = \mathsf{pm}(S)$ puede contener más de una ligadura debido a dos posibles causas:

- Una creación de proceso bloqueada puede estar esperando a que más de una dependencia libre sea evaluada.
- Una ligadura está bloqueada en un canal, el cual tiene asociado un valor *whnf*. En este caso es posible que sean varias las dependencias libres no resueltas de este valor.

Pero el conjunto siempre es finito, pues si el sistema es finito el número de ligaduras activas en él también lo es.

La semántica mínima se obtiene definiendo $\mathcal{LD}(H_p) = H_p \cap \mathcal{LD}_{min}$ para cada $\langle p, H_p\rangle \in S$. Con esta definición se verifica la condición de terminación de la semántica mínima: la variable *main* estará inactiva y tendrá asociado un valor *whnf*, lo que equivale a decir que $\mathsf{pm}(S) = \emptyset$.

Obsérvese que en esta semántica los procesos pueden ser creados especulativamente, pero su cuerpo no será evaluado si su salida no es demandada.

Semántica máxima

En este caso, en cada paso evolucionan en paralelo todas las ligaduras activas en el sistema. Formalmente: si H es un *heap*, el conjunto de sus ligaduras activas se define como $H^A = \{\theta \overset{A}{\mapsto} E \,|\, \theta \in dom(H)\}$, y en la semántica máxima $\mathcal{LD}_{max}(H) = H^A$. De nuevo el número de ligaduras activas del sistema es finito por ser éste finito.

Considerando que la expresión inicial está bien formada, cuando una ligadura está activa existirá una regla local que permitirá evolucionar a dicha ligadura. Este hecho se formaliza en la siguiente proposición:

Proposición 6.8 *Sean $E_0 \notin Val$ una expresión cerrada bien formada, y $\langle p_0, \{main \overset{A}{\mapsto} E_0\}\rangle \Longrightarrow^+ (S, \langle p, H + \{\theta \overset{A}{\mapsto} E\}\rangle)$ una secuencia de configuraciones con al menos un paso. Entonces, si E no es una* #*-expresión, existirá H' tal que $H : \theta \overset{A}{\mapsto} E \longrightarrow H'$.*

Demostración P.6.8
Por inducción estructural sobre E.

- $E = \theta'$: como existen dos tipos de variables, hemos de distinguir dos casos:
  - $\theta' = x$: entonces tendremos $x \overset{\alpha}{\mapsto} E_1$. Si $E_1 = W$, entonces $\alpha = I$ pues el valor habrá sido obtenido en el paso global anterior y la ligadura habrá sido desactivada. Por ello, es posible aplicar la regla VALUE. Si E_1 no es un valor *whnf*, entonces se aplica la regla DEMAND. Finalmente, en el supuesto de que $\theta = \theta' = x$, se aplica a esta ligadura la regla BLACKHOLE.
  - $\theta' = ch$: este supuesto no es posible porque una ligadura que asocia una variable con un canal solamente puede ser una ligadura-esperando-canal (ver Notación 6.3), y, por ende, bloqueada y no activa.
- Si E es una abstracción sucede lo mismo que cuando se suponía era un canal: no puede existir una ligadura del tipo $\theta \overset{A}{\mapsto} E$ (para una mayor aclaración véase el Ejemplo 6.19).
- $E = \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E'$: se puede aplicar la regla LET.
- $E = x\,y$: se contemplan dos posibilidades:
  - Que $x \overset{I}{\mapsto} \backslash x.E_1$, con lo que se puede aplicar β-REDUCTION.
  - Que $x \overset{IAB}{\mapsto} E'$ y E' no sea una abstracción, en cuyo caso se demanda la evaluación de x, es decir, se aplica la regla APP-DEMAND.

Examinados todos los casos posibles de expresión hemos visto que en todos ellos es posible aplicar una regla local.

c.q.d.

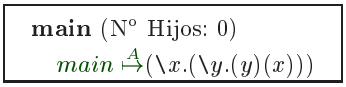
En el enunciado de la Proposición 6.8 se exige al menos un paso en la secuencia de configuraciones. Este requerimiento es necesario porque si E_0 es una abstracción entonces no existe una regla local aplicable, simplemente se desactiva la ligadura de *main* y se termina el cómputo:

Ejemplo 6.19 *La expresión principal es una abstracción.*

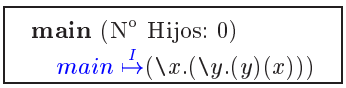
Consideremos la expresión:

$$\backslash x.(\backslash y.y\,x)$$

El sistema inicial es:



La ejecución consiste en un único paso global en el que no se aplican reglas locales, tan solo la global WHNF DEACTIVATION, obteniéndose la siguiente configuración (final):



□

En cualquier otro caso en el que E_0 es una expresión cerrada bien formada, se aplica la Proposición 6.8.

En casi todos los lemas, proposiciones y corolarios enunciados y demostrados hasta ahora se requiere que el sistema bajo consideración sea finito. Para que todos los resultados encajen es necesario que una transición $\Longrightarrow$ transforme estados finitos en estados finitos. Una transición de este tipo se descompone en la forma:

$$\begin{aligned}\Longrightarrow &= \overset{par}{\Longrightarrow} ; \overset{sys}{\Longrightarrow} \\ &= \overset{par}{\Longrightarrow} ; \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{Unbl}{\Longrightarrow} \\ &= \overset{par}{\Longrightarrow} ; \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{wUnbl}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow}\end{aligned}$$

El sistema inicial de una configuración tiene una única ligadura, la de la variable *main*. Es obligatorio entonces demostrar que todas las componentes de $\Longrightarrow$, partiendo de un sistema finito, nos devuelven un sistema finito.

Proposición 6.9 *Sea S un sistema finito. Si $S \overset{\dagger}{\Longrightarrow} S'$, entonces el sistema S' es finito, donde $\dagger$ se ejemplariza como (1) par, (2) comm, (3) pc y (4) Unbl.*

Demostración P.6.9
En la Sección A.5.

A continuación se exponen dos ejemplos en los que una misma expresión se evalua aplicando primero las normas de la semántica mínima, y luego las de la semántica máxima.

Ejemplo 6.20 *Semántica mínima.*

Consideremos la expresión:

$$\texttt{let } x = (y)\#(\backslash t.t), y = (\backslash z.z) \texttt{ in } y$$

que tras ser normalizada e incorporada al sistema inicial produce como resultado:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = x_1, x_0 = (\backslash t.t), x_1 = (y)\#(x_0), y = (\backslash z.z) \texttt{ in } y$

El primer paso de la evaluación es la aplicación de la regla LET con la consiguiente incorporación de ligaduras en el *heap*:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} x_4$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$
$x_4 \overset{I}{\mapsto} (x_5)\#(x_3)$
$x_5 \overset{I}{\mapsto} (\backslash z.z)$

La existencia de un proceso en el nivel superior obliga a su creación, aunque, como veremos, el proceso no será finalmente evaluado.

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{A}{\mapsto} x_5$	$x_6 \overset{B}{\mapsto} ch_7$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} (x_{11})(x_6)$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	
$ch_7 \overset{A}{\mapsto} x_3$	

La primera iteración de evaluación ha finalizado. La siguiente evolución local aplica la regla VALUE a la variable *main*:

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{A}{\mapsto} (\backslash z.z)$	$x_6 \overset{B}{\mapsto} ch_7$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} (x_{11})(x_6)$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	
$ch_7 \overset{A}{\mapsto} x_3$	

La variable *main* ya se encuentra activa y ligada a un valor *whnf*, por lo que se procede a su desactivación:

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{I}{\mapsto} (\backslash z.z)$	$x_6 \overset{B}{\mapsto} ch_7$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} (x_{11})(x_6)$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	
$ch_7 \overset{A}{\mapsto} x_3$	

Con lo que se da por finalizada la evaluación bajo la semántica mínima.

□

Veamos ahora la evaluación según la semántica máxima.

Ejemplo 6.21 *Semántica máxima.*

Recordemos la expresión a evaluar:

$$\texttt{let } x = (y)\#((\backslash t.t)), y = (\backslash z.z) \texttt{ in } y$$

que tras ser normalizada e incorporada al sistema inicial produce como resultado:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x = x_1, x_0 = (\backslash t.t), x_1 = (y)\#(x_0), y = (\backslash z.z) \texttt{ in } y$

La evaluación comienza, al igual que en el caso de la semántica mínima, introduciendo ligaduras por aplicación de la regla LET:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} x_5$
$x_2 \overset{I}{\mapsto} x_4$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$
$x_4 \overset{I}{\mapsto} (x_5)\#(x_3)$
$x_5 \overset{I}{\mapsto} (\backslash z.z)$

Así mismo, se crea el proceso de nivel superior:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{A}{\mapsto} x_5$	$x_6 \overset{B}{\mapsto} ch_7$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} (x_{11})(x_6)$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	
$ch_7 \overset{A}{\mapsto} x_3$	

Ahora evolucionan todas las ligaduras activas, con lo que se aplica VALUE a $main$ y ch_7, y β-REDUCTION a ch_8:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{A}{\mapsto} (\backslash z.z)$	$x_6 \overset{B}{\mapsto} ch_7$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} x_6$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	
$ch_7 \overset{A}{\mapsto} (\backslash t.t)$	

Ahora puede llevarse a cabo la comunicación a través del canal ch_7:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{A}{\mapsto} (\backslash z.z)$	$x_6 \overset{A}{\mapsto} (\backslash x_{12}.x_{12})$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} x_6$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	

Obsérvese la desaparición del canal ch_7. A continuación se procede a desactivar las ligaduras de $main$ y x_6:

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{I}{\mapsto} (\backslash z.z)$	$x_6 \overset{I}{\mapsto} (\backslash x_{12}.x_{12})$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} x_6$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	

Aún a pesar de que la variable *main* está ligada a un valor *whnf*, el cómputo no se da por finalizado, pues la semántica máxima continúa hasta que no haya ligaduras activas en el sistema. Como x_6 está inactiva y ligada al valor *whnf* $\backslash x_{12}.x_{12}$, puede aplicarse la regla VALUE a ch_8:

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{I}{\mapsto} (\backslash z.z)$	$x_6 \overset{I}{\mapsto} (\backslash x_{12}.x_{12})$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	$ch_8 \overset{A}{\mapsto} (\backslash x_{12}.x_{12})$
$x_4 \overset{B}{\mapsto} ch_8$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	

Ahora es posible comunicar a través de ch_8:

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{I}{\mapsto} (\backslash z.z)$	$x_6 \overset{I}{\mapsto} (\backslash x_{12}.x_{12})$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	
$x_4 \overset{A}{\mapsto} (\backslash x_{14}.x_{14})$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	

El canal ch_8 ha cumplido su misión y ha desaparecido. Nótese que esta comunicación era el objetivo de la continuación del cómputo. Resta únicamente desactivar la ligadura de x_4:

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{I}{\mapsto} (\backslash z.z)$	$x_6 \overset{I}{\mapsto} (\backslash x_{12}.x_{12})$
$x_2 \overset{I}{\mapsto} x_4$	$x_{11} \overset{I}{\mapsto} (\backslash x_9.x_9)$
$x_3 \overset{I}{\mapsto} (\backslash t.t)$	
$x_4 \overset{I}{\mapsto} (\backslash x_{14}.x_{14})$	
$x_5 \overset{I}{\mapsto} (\backslash z.z)$	

No quedando ligaduras activas en el sistema, el cómputo ha finalizado.

□

Obsérvese que aún bajo la semántica máxima el tener todas las ligaduras del sistema inactivas no es la única forma de dar por finalizado un cómputo. Puede que se tengan ligaduras bloqueadas derivadas de interbloqueos, como los de los Ejemplos 6.2, 6.3 y 6.8.

Damos aquí por finalizada la presentación de la *semántica operacional con copia evaluada* para Jauja básico. En la siguientes secciones iremos incorporando *streams*, no-determinismo y canales dinámicos a Jauja básico.

6.3. Comunicación vía *streams*

Ahora vamos a añadir al subconjunto de Jauja, del que hemos definido su semántica en la sección anterior, las construcciones necesarias para tratar con *streams* en los canales de comunicación, es decir, vamos a incorporar

E	$::=$	$\Lambda[x_1 : x_2].E_1[\!]E_2$	Λ-abstracción
	$\mid$	L	lista
L	$::=$	`nil`	lista vacía
	$\mid$	$[x_1 : x_2]$	lista no vacía

donde en la última cláusula hemos utilizado ya la forma restringida.

En lo sucesivo denotaremos por *List* al conjunto de listas, descrito por medio de la categoría sintáctica L.

6.3.1. Transiciones locales

La evaluación realizada hasta ahora exige que la copia de variables de un *heap* a otro solamente se lleve a cabo cuando éstas se encuentren ligadas a valores *whnf*. Esta imposición hace que las variables libres de una expresión a comunicar, que estará en *whnf*, estén evaluadas, con lo que dicha expresión estará casi en forma normal, en el sentido de que todas sus variables libres están ya evaluadas, con lo que bastaría con realizar las sustituciones oportunas para obtener un valor en forma normal según la definición dada en el glosario (Apéndice B). En consecuencia, es necesario añadir a la serie de transiciones locales en la Figura 6.2 las necesarias para manejar *streams*. Estas nuevas reglas se concentran en la obtención de una Λ-abstracción y su aplicación, y en la demanda de evaluación de la cabeza de un *stream*.

6.3.1.1. Λ-abstracción

El tratamiento de las Λ-abstracciones es similar al de las λ-abstracciones, salvo que hay que introducir otro punto de demanda, debido al carácter estricto de la aplicación que tenemos ahora. Además, la B-reducción, o aplicación, ha de distinguir entre el caso de lista vacía y el de no vacía. Las correspondientes reglas se muestran en la Figura 6.7.

La estrictez en la aplicación de una Λ-abstracción se traduce en la incorporación de la regla Λ-DEMAND: una vez que x, la variable de la abstracción, está ligada a una Λ-abstracción, si la variable argumento y no está asociada a un valor, entonces hay que

(Λ-demand)
si $E \notin Val$
$$H + \{x \overset{I}{\mapsto} \Lambda[y_1 : y_2].E_1 \| E_2, y \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} x\, y \longrightarrow H + \{x \overset{I}{\mapsto} \Lambda[y_1 : y_2].E_1 \| E_2, y \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} x\, y\}$$

(B-reduction empty list)
$$H + \{x \overset{I}{\mapsto} \Lambda[y_1 : y_2].E_1 \| E_2, z \overset{I}{\mapsto} \texttt{nil}\} : \theta \overset{A}{\mapsto} x\, z \longrightarrow H + \{x \overset{I}{\mapsto} \Lambda[y_1 : y_2].E_1 \| E_2, z \overset{I}{\mapsto} \texttt{nil}, \theta \overset{A}{\mapsto} E_1\}$$

(B-reduction non-empty list)
$$H + \{x \overset{I}{\mapsto} \Lambda[y_1 : y_2].E_1 \| E_2, z \overset{I}{\mapsto} [z_1 : z_2]\} : \theta \overset{A}{\mapsto} x\, z \longrightarrow$$
$$\longrightarrow H + \{x \overset{I}{\mapsto} \Lambda[y_1 : y_2].E_1 \| E_2, z \overset{I}{\mapsto} [z_1 : z_2], \theta \overset{A}{\mapsto} E_2[z_1/y_1, z_2/y_2]\}$$

Figura 6.7: Jauja básico con *streams*: reglas locales de Λ-abstracción

obtenerlo. Para ello se bloquea la aplicación y se activa, si procede, la evaluación de y, de manera análoga a como lo hacían las reglas DEMAND y APP-DEMAND de la Figura 6.2.

Cuando dicho valor haya sido obtenido, se podrá proceder a realizar la B-reducción. Ésta viene definida por dos reglas:

B-REDUCTION EMPTY LIST : lleva a cabo la B-reducción cuando el argumento está ligado a la lista vacía, `nil`, en cuyo caso la evaluación ha de continuar con E_1.

B-REDUCTION NON-EMPTY LIST : desarrolla la B-reducción en el caso de que el argumento no sea la lista vacía. La aplicación sustituye en la expresión E_2 los parámetros formales y_1 e y_2 por los actuales z_1 y z_2, respectivamente, continuándose con la evaluación de la expresión así obtenida.

Ahora se tienen dos tipos de aplicaciones; sin embargo, es la regla APP-DEMAND la que introduce la demanda para obtener la abstracción en ambos casos.

Ejemplo 6.22 *Aplicación de listas: demanda y B-reducción de una lista vacía.*

$$\texttt{let}\ x_0 = \backslash y.y, x_1 = (\Lambda[y_1 : y_2].x_2 \| [y_1 : y_2]), x_2 = 1\ \texttt{in}\ (x_0\, x_1)(\texttt{nil})$$

Tras normalizar esta expresión su evaluación procede de la siguiente forma:

$$\begin{array}{l} main \overset{A}{\mapsto} \texttt{let}\ x_0 = \backslash y.y, \\ \qquad\qquad\quad x_1 = (\Lambda[y_1 : y_2].x_2 \| [y_1 : y_2]), \\ \qquad\qquad\quad x_2 = 1, \\ \qquad\qquad\quad x_3 = \texttt{let}\ x_4 = x_0\, x_1, \\ \qquad\qquad\qquad\qquad\ x_5 = \texttt{nil}, \\ \qquad\qquad\qquad\qquad\ x_6 = x_4\, x_5 \\ \qquad\qquad\qquad\ \texttt{in}\ x_6 \\ \qquad\qquad \texttt{in}\ x_3 \end{array}$$

La primera regla que se aplica, LET, incorpora en el *heap* las nuevas variables locales, pero renombradas:

LET

$main \overset{A}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{I}{\mapsto} \texttt{let } x_4 = x_7\ x_8, x_5 = \texttt{nil}, x_6 = x_4\ x_5 \texttt{ in } x_6$

A continuación, se demanda la evaluación de la variable a la que se liga *main*:

DEMAND

$main \overset{B}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{A}{\mapsto} \texttt{let } x_4 = x_7\ x_8, x_5 = \texttt{nil}, x_6 = x_4\ x_5 \texttt{ in } x_6$

Los siguientes pasos son la evaluación de otra declaración local y la demanda para evaluar la variable que es su cuerpo:

LET

$main \overset{B}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{A}{\mapsto} x_{13}$
$x_{11} \overset{I}{\mapsto} x_7\ x_8$
$x_{12} \overset{I}{\mapsto} \texttt{nil}$
$x_{13} \overset{I}{\mapsto} x_{11}\ x_{12}$

DEMAND

$main \overset{B}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{B}{\mapsto} x_{13}$
$x_{11} \overset{I}{\mapsto} x_7\ x_8$
$x_{12} \overset{I}{\mapsto} \texttt{nil}$
$x_{13} \overset{A}{\mapsto} x_{11}\ x_{12}$

En este momento no se puede llevar a cabo la reducción $x_9\ x_{10}$, por lo que se procede a demandar la evaluación de x_9:

APP-DEMAND

$main \overset{B}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{B}{\mapsto} x_{13}$
$x_{11} \overset{A}{\mapsto} x_7\ x_8$
$x_{12} \overset{I}{\mapsto} \texttt{nil}$
$x_{13} \overset{B}{\mapsto} x_{11}\ x_{12}$

β-REDUCTION

$main \overset{B}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{B}{\mapsto} x_{13}$
$x_{11} \overset{A}{\mapsto} x_8$
$x_{12} \overset{I}{\mapsto} \texttt{nil}$
$x_{13} \overset{B}{\mapsto} x_{11}\ x_{12}$

VALUE

$main \overset{B}{\mapsto} x_{10}$
$x_7 \overset{I}{\mapsto} \backslash y.y$
$x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_9 \overset{I}{\mapsto} 1$
$x_{10} \overset{B}{\mapsto} x_{13}$
$x_{11} \overset{A}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])$
$x_{12} \overset{I}{\mapsto} \texttt{nil}$
$x_{13} \overset{B}{\mapsto} x_{11}\ x_{12}$

Tras desbloquear y desactivar, ya se puede llevar a cabo la B-reducción de $x_9\ x_{10}$:

B-REDUCTION EMPTY LIST

$$
\begin{array}{l}
main \overset{B}{\mapsto} x_{10} \\
x_7 \overset{I}{\mapsto} \backslash y.y \\
x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2])
\end{array}
$$

$$
\begin{array}{l}
x_9 \overset{I}{\mapsto} 1 \\
x_{10} \overset{B}{\mapsto} x_{13} \\
x_{11} \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2]) \\
x_{12} \overset{I}{\mapsto} \texttt{nil} \\
x_{13} \overset{A}{\mapsto} x_9
\end{array}
$$

Desbloqueos, desactivaciones y la aplicación de la regla VALUE dan como resultado:

$$
\begin{array}{l}
main \overset{I}{\mapsto} 1 \\
x_7 \overset{I}{\mapsto} \backslash y.y \\
x_8 \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2]) \\
x_9 \overset{I}{\mapsto} 1 \\
x_{10} \overset{I}{\mapsto} 1 \\
x_{11} \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2]) \\
x_{12} \overset{I}{\mapsto} \texttt{nil} \\
x_{13} \overset{I}{\mapsto} 1
\end{array}
$$

□

Ejemplo 6.23 *Aplicación de listas: B-reducción de una lista no vacía.*

$$\texttt{let } x_0 = 3, x_1 = \texttt{nil}, x_2 = 1 \texttt{ in } ((\Lambda[y_1 : y_2].x_2 \llbracket [y_1 : y_2])([x_0 : x_1])$$

La normalización de esta expresión se liga a la variable *main* del proceso inicial:

$$
\begin{array}{l}
main \overset{A}{\mapsto} \texttt{let } x_0 = 3, \\
\qquad\qquad\qquad\quad x_1 = \texttt{nil}, \\
\qquad\qquad\qquad\quad x_2 = 1, \\
\qquad\qquad\qquad\quad x_3 = \texttt{let } x_4 = (\Lambda[y_1 : y_2].x_2 \llbracket [y_1 : y_2]), \\
\qquad\qquad\qquad\qquad\qquad\quad x_5 = ([x_0 : x_1]), \\
\qquad\qquad\qquad\qquad\qquad\quad x_6 = x_4\, x_5 \\
\qquad\qquad\qquad\qquad\quad \texttt{in } x_6 \\
\qquad\qquad\quad \texttt{in } x_3
\end{array}
$$

El primer paso incorpora al *heap* las nuevas ligaduras:

LET

$$
\begin{array}{l}
main \overset{A}{\mapsto} x_{10} \\
x_7 \overset{I}{\mapsto} 3 \\
x_8 \overset{I}{\mapsto} \texttt{nil} \\
x_9 \overset{I}{\mapsto} 1 \\
x_{10} \overset{I}{\mapsto} \texttt{let } x_4 = (\Lambda[y_1 : y_2].x_9 \llbracket [y_1 : y_2]), x_5 = ([x_7 : x_8]), x_6 = x_4\, x_5 \texttt{ in } x_6
\end{array}
$$

La variable *main* se bloquea porque x_{10} no está ligada a un valor *whnf*:

DEMAND

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_7 \overset{I}{\mapsto} 3 \\ x_8 \overset{I}{\mapsto} \texttt{nil} \\ x_9 \overset{I}{\mapsto} 1 \\ x_{10} \overset{A}{\mapsto} \texttt{let } x_4 = (\Lambda[y_1 : y_2].x_9[\![y_1 : y_2]\!]), x_5 = ([x_7 : x_8]), x_6 = x_4\, x_5 \texttt{ in } x_6 \end{array}$$

La evaluación de x_{10} hace que se introduzcan nuevas variables en el *heap* y produce la posterior demanda de la evaluación de x_{13}:

LET

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_7 \overset{I}{\mapsto} 3 \\ x_8 \overset{I}{\mapsto} \texttt{nil} \\ x_9 \overset{I}{\mapsto} 1 \\ x_{10} \overset{A}{\mapsto} x_{13} \\ x_{11} \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9[\![y_1 : y_2]\!]) \\ x_{12} \overset{I}{\mapsto} [x_7 : x_8] \\ x_{13} \overset{I}{\mapsto} x_{11}\, x_{12} \end{array}$$

DEMAND

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_7 \overset{I}{\mapsto} 3 \\ x_8 \overset{I}{\mapsto} \texttt{nil} \\ x_9 \overset{I}{\mapsto} 1 \\ x_{10} \overset{B}{\mapsto} x_{13} \\ x_{11} \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9[\![y_1 : y_2]\!]) \\ x_{12} \overset{I}{\mapsto} [x_7 : x_8] \\ x_{13} \overset{A}{\mapsto} x_{11}\, x_{12} \end{array}$$

En el siguiente paso se lleva a cabo la B-reducción de $x_{11}\, x_{12}$. Como la variable x_{12} está ligada a una lista no vacía, se aplica la B-reducción correspondiente a este caso. En concreto el caso de la Λ-abstracción con la que continúa la evaluación es el segundo, $[y_1 : y_2]$, que al aplicar la abstracción sobre x_{12} nos conduce a $[x_7 : x_8]$.

B-REDUCTION NON-EMPTY LIST

$$\begin{array}{l} main \overset{B}{\mapsto} x_{10} \\ x_7 \overset{I}{\mapsto} 3 \\ x_8 \overset{I}{\mapsto} \texttt{nil} \\ x_9 \overset{I}{\mapsto} 1 \\ x_{10} \overset{B}{\mapsto} x_{13} \\ x_{11} \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9[\![y_1 : y_2]\!]) \\ x_{12} \overset{I}{\mapsto} [x_7 : x_8] \\ x_{13} \overset{A}{\mapsto} [x_7 : x_8] \end{array}$$

Tras varios desbloqueos, desactivaciones y aplicaciones de VALUE, se obtiene:

B-REDUCTION NON-EMPTY LIST

$$
\begin{array}{l}
main \overset{I}{\mapsto} [x_7 : x_8] \\
x_7 \overset{I}{\mapsto} 3 \\
x_8 \overset{I}{\mapsto} \texttt{nil} \\
x_9 \overset{I}{\mapsto} 1 \\
x_{10} \overset{I}{\mapsto} [x_7 : x_8] \\
x_{11} \overset{I}{\mapsto} (\Lambda[y_1 : y_2].x_9 \| [y_1 : y_2]) \\
x_{12} \overset{I}{\mapsto} [x_7 : x_8] \\
x_{13} \overset{I}{\mapsto} [x_7 : x_8]
\end{array}
$$

□

Las reglas locales que hemos introducido en la Figura 6.7 no cambian el número de ligaduras del sistema, por lo que transforman un sistema finito en otro que también lo es.

Estas reglas locales expresan cómo se tratan las listas de Jauja. La inclusión de listas fue consecuencia del deseo de incluir *streams* en el lenguaje. Pero listas y *streams* tienen comportamientos bien distintos, lo cual queda reflejado en la diferente evaluación que se realiza cuando una lista adquiere la categoría de *stream*.

6.3.1.2. Demanda de un *stream*

Una lista es considerada *stream* cuando está asociada a un canal de comunicación. El proceso de evaluación difiere, pues una lista ordinaria es perezosa mientras que el *stream* no lo es: cuando se evalúa un *stream* su cabeza ha de ser reducida hasta que se convierta en un *valor comunicable* (forma normal, entendiendo como tal un valor *whnf* en el que todas sus dependencias libres han sido resueltas), mientras que en el caso de una lista ordinaria no se fuerza la evaluación completa de ningún elemento de la lista, salvo demanda explícita de la misma.

La regla de la Figura 6.8 rige la demanda en el caso de los *streams*:

(*stream* demand)
si $E \notin Val$

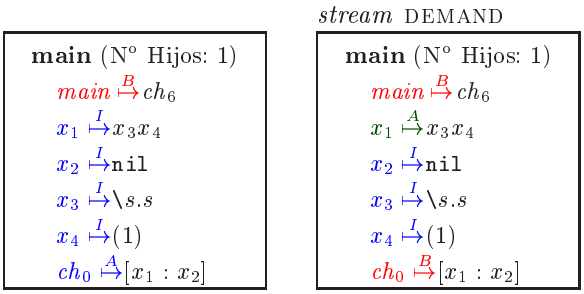
$$H + \{x_1 \overset{IAB}{\mapsto} E\} : ch \overset{A}{\mapsto} [x_1 : x_2] \longrightarrow H + \{x_1 \overset{AAB}{\mapsto} E, ch \overset{B}{\mapsto} [x_1 : x_2]\}$$

Figura 6.8: Jauja básico con *streams*: regla local de demanda

La evaluación ha asociado al canal *ch* un valor en *whnf*. Sin embargo, esto no basta, pues no es esta lista la que enviará directamente, sino sus elementos uno por uno. Es por ello que se bloquea la ligadura del canal y se demanda la evaluación de la cabeza, siguiendo el proceso habitual de demanda.

Ejemplo 6.24 *Demanda de la evaluación de un stream.*

El siguiente *heap* contiene una ligadura, la de ch_0, que necesita la evaluación de x_1, su cabeza. Esta última se evaluará hasta *whnf*:



Será la regla HEAD-*stream* COMMUNICATION DEMAND, que veremos en la Figura 6.11, la que permita la terminación del proceso de evaluación a forma normal de la cabeza del *stream*.

□

Al igual que sucedía con las reglas de la sección anterior, la incluida en la Figura 6.8 tampoco varía el número de ligaduras del sistema, por lo que si éste es finito se transforma en otro que también lo es.

La evaluación local relacionada con los *streams* ha quedado ya definida, pero la inclusión de esta construcción en Jauja hace necesarios también algunos cambios en la evolución global del sistema.

6.3.2. Transiciones globales

Las transiciones globales comprenden las tareas de creación de procesos, comunicación y reetiquetado de ligaduras por motivo de desbloqueo, demanda o desactivación.

6.3.2.1. Creación de procesos

La regla de creación de procesos permanece invariable, aún habiendo sido incluidos los *streams*, pues el que un canal vaya o no a ser un *stream* es irrelevante a la hora de establecer el *heap* inicial del nuevo proceso.

6.3.2.2. Comunicación

La comunicación es una de las reglas globales que más cambios sufre tras esta incorporación. Las nuevas reglas quedan recogidas en la Figura 6.9.

(value communication)

$$\text{si } \mathtt{nf}(W, H_p) = \emptyset \wedge (W \in \mathit{List} \Rightarrow W \equiv \mathtt{nil})$$
$$\mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{vCom}{\longrightarrow}$$
$$\overset{vCom}{\longrightarrow} (S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathtt{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} \eta(W)\}\rangle)$$

(head-*stream* communication)

$$H'_p = H_p + \{ch \overset{\alpha}{\mapsto} [x_1 : x_2], x_1 \overset{I}{\mapsto} W\}$$
$$\text{si } (\mathtt{nf}(W, H'_p) = \emptyset) \wedge (\mathsf{isB}(x_1, H'_p))$$
$$\mathit{fresh}(y_1, y_2), \mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} [x_1 : x_2], x_1 \overset{I}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{sCom}{\longrightarrow}$$
$$\overset{sCom}{\longrightarrow} (S, \langle p, H_p + \{ch \overset{A}{\mapsto} x_2, x_1 \overset{I}{\mapsto} W\}\rangle, \langle c, H_c + \eta(\mathtt{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} [y_1 : y_2], y_1 \overset{A}{\mapsto} \eta(W), y_2 \overset{B}{\mapsto} ch\}\rangle)$$

Figura 6.9: Jauja básico con *streams*: reglas globales de comunicación

El planteamiento inicial de la regla de comunicación VALUE COMMUNICATION varía con respecto a la de la Sección 6.2.2.2, pues ahora los valores que se comunican ya no son sólo *whnf*, sino que tienen que ser valores comunicables, i.e. estar en forma normal *del todo* cuando se trate de listas. No obstante, este cambio no se ve reflejado (por encontrarnos en el enfoque de copia evaluada) pues la llamada a la función `nf` controla ambas condiciones. De la misma forma que sucedía en el caso de canales de único valor, el canal-*stream* de comunicación desaparece cuando se comunica `nil`, ya que su cometido ha sido cumplido. Además, esta regla recoge el caso de comunicación del final de un *stream*, pues si *ch* se encuentra ligado a `nil`, se comunica este valor y se cierra el canal.

En cuanto a la comunicación de la cabeza de un *stream*, llevada a cabo por la regla HEAD-*stream* COMMUNICATION, dicha cabeza tiene que ser un valor comunicable, lo que significa que, si es una lista, cada componente suya tiene que estar en forma normal, es decir, tiene que estar lista para ser comunicada. De nuevo esto es equivalente a que todas sus dependencias libres estén resueltas. Los cambios que se producen en el sistema derivados de esta comunicación son:

- En el *heap* del proceso p, el canal ch se liga a x_2.
- En el proceso consumidor:
  - La variable que estaba bloqueada en ch se activa y se liga a una nueva lista, $[y_1 : y_2]$, definida por sendas variables frescas, tanto la cabeza como para la cola.
  - La variable cabeza se liga a W, convenientemente renombrado.
  - La variable cola se liga a ch.
  - Se copian en este *heap* las dependencias libres de W.

Además, si la cabeza a comunicar es una lista, ésta tiene que estar bien construida, es decir, tiene que acabar en `nil`; y si contiene listas como elementos, también han de verificar esta condición. Para ello definimos las funciones siguientes:

$$\mathsf{lisB}(x,H) = \begin{cases} \mathsf{lisB}(x_1,H') \wedge \mathsf{lisB}(x_2,H') \wedge \mathsf{lisBC}(x_2,H'), & \text{si } x \overset{\alpha}{\mapsto} [x_1 : x_2] \in H; \\ \quad \texttt{donde } H' = H \setminus \{x \overset{\alpha}{\mapsto} [x_1 : x_2]\} & \\ \text{true}, & \text{e.o.c.} \end{cases}$$

$$\mathsf{lisBC}(x,H) = \begin{cases} \mathsf{lisBC}(x_2,H'), & \text{si } x \overset{\alpha}{\mapsto} [x_1 : x_2] \in H; \\ \quad \texttt{donde } H' = H \setminus \{x \overset{\alpha}{\mapsto} [x_1 : x_2]\} & \\ \text{true}, & \text{si } x \overset{\alpha}{\mapsto} \texttt{nil} \in H; \\ \text{false}, & \text{e.o.c.} \end{cases}$$

De ellas, lisB comprueba que ambas componentes sean buenas lista, para lo cual recurre a lisBC para exigir que la cola termine en `nil`.

Vistas estas reglas, podemos definir formalmente las condiciones que indican cuándo un valor se puede comunicar:

Definición 6.9 Sea H_p un *heap* en el que se incluye o bien la ligadura $ch \overset{\alpha}{\mapsto} W$ o bien las ligaduras $ch \overset{\alpha}{\mapsto} [x_1 : x_2], x_1 \overset{I}{\mapsto} W$. Decimos que W es un *valor comunicable* en el primer caso si $(\mathsf{nf}(W, H_p) = \emptyset \wedge (W \in \mathit{List} \Rightarrow W \equiv \texttt{nil}))$, y en el segundo caso si $(\mathsf{nf}(W, H_p) = \emptyset) \wedge (\mathsf{lisB}(x_1, H_p))$.

□

Bajo estas condiciones, la comunicación de la cabeza procede, asociando a la variable del receptor —que estará bloqueada en el canal— una lista, cuya cabeza queda ligada a la cabeza comunicada, y cuya cola queda en espera de la comunicación del resto del *stream*, a través del mismo canal. Además, la función nh recolecta todas las variables libres en la cabeza comunicada. Tanto la función nh como la función nf siguen estando definidas según se indicó en la Figura 6.4. Sin embargo, la función `dexp` tiene que ser extendida para recoger las nuevas construcciones sintácticas relacionadas con las listas. Las modificaciones precisas se exponen en la Figura 6.10.

$$\begin{aligned} \texttt{dexp}(\texttt{nil}) &= \emptyset \\ \texttt{dexp}(\Lambda[x_1, x_2].E_1 [\![E_2) &= \{E_1, E_2\} \\ \texttt{dexp}([E_1 : E_2]) &= \{E_1, E_2\} \end{aligned}$$

Figura 6.10: Jauja básico con *streams*: detección de dependencias

Veamos ejemplos de ambos tipos de comunicación:

Ejemplo 6.25 *Comunicación de un valor simple.*

En el *heap* que aparece a continuación, ch_0 está ligada al cierre de un *stream*.

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{B}{\mapsto} ch_0$
$x_6 \overset{B}{\mapsto} ch_8$	$ch_8 \overset{B}{\mapsto} x_7$
$ch_0 \overset{I}{\mapsto} \texttt{nil}$	

Se procede a la comunicación. Como el cometido de ch_0 ha sido cumplido, desaparece del sistema:

VALUE COMMUNICATION

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{A}{\mapsto} \texttt{nil}$
$x_6 \overset{B}{\mapsto} ch_8$	$ch_8 \overset{B}{\mapsto} x_7$

A continuación, tras el desbloqueo de ch_8 y la desactivación de x_7, se aplica la regla VALUE:

VALUE

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{I}{\mapsto} \texttt{nil}$
$x_6 \overset{B}{\mapsto} ch_8$	$ch_8 \overset{A}{\mapsto} \texttt{nil}$

Y llegamos a la otra comunicación de cierre de *stream*, lo que provoca la desaparición del canal ch_8:

VALUE

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{I}{\mapsto} \texttt{nil}$
$x_6 \overset{A}{\mapsto} \texttt{nil}$	

Desbloqueos, desactivaciones y la regla VALUE hacen el resto:

VALUE

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{I}{\mapsto} \texttt{nil}$	$x_7 \overset{I}{\mapsto} \texttt{nil}$
$x_6 \overset{I}{\mapsto} \texttt{nil}$	

□

Ejemplo 6.26 *Comunicación de la cabeza de un stream.*

En el siguiente *heap*, ch_0 está ligada a un *stream* cuya cabeza es un valor en forma normal que se puede comunicar:

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{B}{\mapsto} ch_0$
$x_1 \overset{I}{\mapsto} (\backslash x.x)$	$ch_8 \overset{B}{\mapsto} x_7$
$x_2 \overset{I}{\mapsto} \texttt{nil}$	
$x_6 \overset{B}{\mapsto} ch_8$	
$ch_0 \overset{I}{\mapsto} [x_1 : x_2]$	

HEAD-*stream* COMMUNICATION

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{A}{\mapsto} [x_9 : x_{10}]$
$x_1 \overset{I}{\mapsto} (\backslash x.x)$	$x_9 \overset{A}{\mapsto} (\backslash x.x)$
$x_2 \overset{I}{\mapsto} \texttt{nil}$	$x_{10} \overset{B}{\mapsto} ch_0$
$x_6 \overset{B}{\mapsto} ch_8$	$ch_8 \overset{B}{\mapsto} x_7$
$ch_0 \overset{A}{\mapsto} x_2$	

□

La comunicación de un valor, tanto cuando se aplica VALUE COMMUNICATION como HEAD-*stream* COMMUNICATION, puede permitir que se realicen otras comunicaciones pendientes. Por ello, las dos reglas de la Figura 6.9 tienen que ser aplicadas intercalada y repetidamente, hasta que no sea posible ninguna otra comunicación. De este modo la regla de paso reiterado para comunicaciones queda como sigue:

$$\overset{Com}{\Longrightarrow} = (\overset{vCom}{\Longrightarrow} ; \overset{sCom}{\Longrightarrow})^*.$$

Demostremos ahora, al igual que ya hemos realizado en ocasiones previas, la no interferencia entre diversas comunicaciones y que esta composición global está bien definida.

Las proposiciones que a continuación se detallan, y que se demuestran en la Sección A.6, establecen la ausencia de interferencias:

Proposición 6.10 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} W_1 \in H_1$ una ligadura en la que W_1 es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} W_2 \in H_2$ una ligadura en la que W_2 es también comunicable, verificándose $ch_1 \neq ch_2$. Entonces la aplicación de $\overset{vCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\overset{vCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.10
En la Sección A.6.

Veamos que tampoco hay interferencia entre las comunicaciones de cabezas de *streams*:

Proposición 6.11 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_2^1] \in H_1$ una ligadura tal que $x_1^1 \overset{I}{\mapsto} W_1 \in H_1 \wedge W_1$ es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} [x_1^2 : x_2^2] \in H_2$ una ligadura tal que $x_1^2 \overset{I}{\mapsto} W_2 \in H_2 \wedge W_2$ es comunicable, verificando $ch_1 \neq ch_2$. Entonces la aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.11
En la Sección A.6.

Para terminar con la demostración de la ausencia de interferencias, veamos que la realización de una comunicación simple no elimina la posibilidad de una comunicación de *stream* que era factible, y viceversa.

Proposición 6.12 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} W_1 \in H_1$ una ligadura en la que W_1 es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} [x_1 : x_2] \in H_2$ una ligadura tal que $x_1 \overset{I}{\mapsto} W_2 \in H_2 \wedge W_2$ es comunicable. Entonces la aplicación de $\overset{vCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.12
En la Sección A.6.

Proposición 6.13 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} [x_1 : x_2] \in H_1$ una ligadura tal que $x_1 \overset{I}{\mapsto} W_1 \in H_1 \wedge W_1$ es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} W_2 \in H_2$ una ligadura tal que W_2 es comunicable. Entonces la aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\overset{vCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.13
En la Sección A.6.

Veamos ahora que $\overset{Com}{\Longrightarrow}$ está bien definida.

Proposición 6.14 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{vCom}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{vCom}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.14
En la Sección A.7.

Demostremos que la otra componente de $\overset{Com}{\Longrightarrow}$ también está bien definida.

La comunicación de la cabeza de un *stream* puede provocar que la variable receptora sea de canal, de modo que entonces se habilite una nueva comunicación a través de dicho canal. Antes de ver un ejemplo, introduzcamos la siguiente notación:

Notación 6.4

- Una *ligadura-canal-stream* es una ligadura de la forma $ch \overset{\alpha}{\mapsto} [x_1 : x_2]$.
- Una *ligadura-canal-bloqueada-canal* es una ligadura de la forma $ch_1 \overset{B}{\mapsto} ch_2$.

□

Ejemplo 6.27 *Comunicación de la cabeza de un stream.*

En el siguiente *heap*, ch_0 está ligado a un *stream* cuya cabeza es un valor en forma normal que se puede comunicar. El receptor de este *stream* es, a su vez, un canal.

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$ch_7 \overset{B}{\mapsto} ch_0$
$x_1 \overset{I}{\mapsto} (\backslash x.x)$	
$x_2 \overset{I}{\mapsto} \texttt{nil}$	
$x_6 \overset{B}{\mapsto} ch_7$	
$ch_0 \overset{I}{\mapsto} [x_1 : x_2]$	

HEAD-*stream* COMMUNICATION

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$ch_7 \overset{A}{\mapsto} [x_9 : x_{10}]$
$x_1 \overset{I}{\mapsto} (\backslash x.x)$	$x_9 \overset{A}{\mapsto} (\backslash x.x)$
$x_2 \overset{I}{\mapsto} \texttt{nil}$	$x_{10} \overset{B}{\mapsto} ch_0$
$x_6 \overset{B}{\mapsto} ch_7$	
$ch_0 \overset{A}{\mapsto} x_2$	

El número de ligaduras-canal-*stream* no se ha reducido, pero sí el de ligaduras-canal-bloqueadas-canal. De nuevo estamos ante la posibilidad de comunicar la cabeza de un *stream* (de hecho la del mismo *stream*) que vuelve al proceso principal.

HEAD-*stream* COMMUNICATION

main (N° Hijos: 1)	$\mathbf{p_1}$ (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$ch_7 \overset{A}{\mapsto} x_{10}$
$x_1 \overset{I}{\mapsto} (\backslash x.x)$	$x_9 \overset{A}{\mapsto} (\backslash x.x)$
$x_2 \overset{I}{\mapsto} \texttt{nil}$	$x_{10} \overset{B}{\mapsto} ch_0$
$x_6 \overset{A}{\mapsto} [x_{11} : x_{12}]$	
$x_{11} \overset{A}{\mapsto} (\backslash x.x)$	
$x_{12} \overset{B}{\mapsto} ch_7$	
$ch_0 \overset{A}{\mapsto} x_2$	

□

Para que sigan teniendo lugar las comunicaciones de cabeza de *stream* han de existir ligaduras-canal-*stream*. Hemos visto que cuando se comunica una cabeza puede suceder que el número de ligaduras-canal-*stream* no decrezca, al haber sido habilitada una nueva comunicación. Sin embargo, la posibilidad de habilitar nuevas comunicaciones de cabeza termina cuando se acaban las ligaduras-canal-bloqueadas-canal, con lo que solamente se pueden realizar las comunicaciones de cabezas de *stream* que ya se tienen. Es por ello que la demostración de que $\overset{sCom}{\Longrightarrow}$ está bien definida pasa por la veracidad del Lema A.26 incluido en la Sección A.8.

Proposición 6.15 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{sCom}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{sCom}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.15
En la Sección A.8.

Finalmente veamos que $\overset{vCom}{\Longrightarrow}$; $\overset{sCom}{\Longrightarrow}$ está bien definida.

Teorema 6.2 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{vCom}{\Longrightarrow}$; $\overset{sCom}{\Longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{Com}{\Longrightarrow}$ para S está bien definida.*

Demostración T.6.2
Condición suficiente para que $\overset{Com}{\Longrightarrow}$ esté bien definida es que la siguiente suma decrezca de manera estricta tras cada iteración.

$$\text{n}^{\text{o}} \text{ ligaduras-esperando-canal} + \text{n}^{\text{o}} \text{ ligaduras-canal-}\textit{stream}$$

Hemos considerado el número de ligaduras-esperando-canal porque entre ellas se encuentran las ligaduras-canal-bloqueadas-canal.

Según indica el Lema A.23 de la Sección A.7, cuando se aplica la regla $\overset{vCom}{\Longrightarrow}$ el número de ligaduras-esperando-canal decrece de manera estricta, y, como los valores comunicados no son listas, el número de ligaduras-canal-*stream* no puede crecer. Atendiendo al resultado del Lema A.26, la suma anterior decrece de manera estricta tras la aplicación de $\overset{sCom}{\Longrightarrow}$. En consecuencia, tras aplicar ambas reglas la suma anterior ha decrecido de manera estricta, de lo que se deduce que la imagen de S por $\overset{Com}{\Longrightarrow}$ está bien definida.
c.q.d.

6.3.2.3. Planificación

En consonancia con la Sección 6.2.2.3, este cambio de estados supone realizar:

1. El desbloqueo de las ligaduras que dependían de una variable ya asociada a valor.
2. La desactivación de las ligaduras que ya han alcanzado una *whnf*.
3. El bloqueo de las creaciones de proceso que no han podido realizarse.
4. La demanda de evaluación de las ligaduras necesarias para llevar a cabo las creaciones de proceso pendientes.
5. La demanda de evaluación de las ligaduras necesarias para llevar a cabo las comunicaciones pendientes, tanto de valores simples como vía *streams*.

Las cuatro primeras tareas son llevadas a cabo por las primeras reglas globales presentadas en la Figura 6.6, salvo que en la regla WHNF DEACTIVATION se impone la condición: $\neg(\theta = ch \wedge \theta \overset{A}{\mapsto} [x_1 : x_2] \wedge x_1 \overset{\alpha}{\mapsto} E \wedge E \notin Val)$ para poder demandar la evaluación de la cabeza del *stream*. La última tarea se realiza por medio de las reglas globales de la Figura 6.11. De ellas, la primera corresponde a la modificación de la regla VALUE COMMUNICATION DEMAND de la Figura 6.6, de la forma que se necesita al haberse realizado la inclusión de *streams*.

(value communication demand)

$$\text{si } W \notin List \wedge x \overset{I}{\mapsto} E \in \mathsf{nf}(W, H)$$
$$(S, \langle p, H + \{ch \overset{I}{\mapsto} W\}\rangle) \overset{vComd}{\longrightarrow} (S, \langle p, H + \{ch \overset{I}{\mapsto} W, x \overset{A}{\mapsto} E\}\rangle)$$

(head-*stream* communication demand)

$$H' = H + \{x_1 \overset{I}{\mapsto} W, ch \overset{I}{\mapsto} [x_1 : x_2]\}$$
$$\text{si } x \overset{I}{\mapsto} E \in \mathsf{nf}(W, H')$$
$$(S, \langle p, H + \{x_1 \overset{I}{\mapsto} W, ch \overset{I}{\mapsto} [x_1 : x_2]\}\rangle) \overset{hsComd}{\longrightarrow} (S, \langle p, H + \{x_1 \overset{I}{\mapsto} W, ch \overset{I}{\mapsto} [x_1 : x_2], x \overset{A}{\mapsto} E\}\rangle)$$

Figura 6.11: Jauja básico con *streams*: reglas globales de reetiquetado

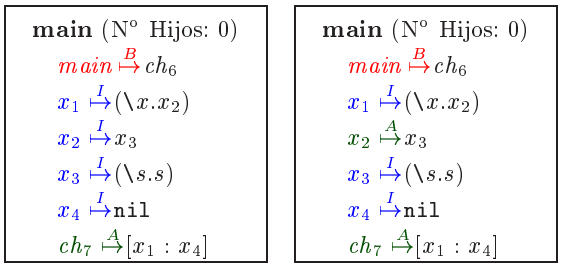
El bloqueo de creaciones de proceso y la demanda producida por este bloqueo permanecen invariables tras la introducción de *streams* en Jauja. Sin embargo, la demanda para comunicaciones varía, pues hay que distinguir entre la comunicación de un valor simple y la comunicación de la cabeza de un *stream*. En el primer caso, se estaría en disposición de comunicar un valor *whnf*, de no ser por las dependencias sin resolver, que VALUE COMMUNICATION DEMAND demanda para evaluar; el caso de cierre de un *stream* no pasa por esta regla, pues `nil` no puede tener dependencias. En el segundo caso, para que la cabeza del *stream* pueda ser comunicada, hay que resolver sus dependencias, que HEAD-*stream* COMMUNICATION DEMAND demanda, alcanzándose así la deseada forma normal comunicable.

Para mostrar cómo funciona la demanda a la hora de comunicar la cabeza de un *stream* recurrimos a un ejemplo similar al Ejemplo 6.16:

Ejemplo 6.28 *Demanda para comunicar la cabeza de un stream.*

En el *heap* detallado seguidamente, x_1 está ligada a un valor en forma normal que se podría comunicar. Sin embargo, x_1 depende de la variable libre x_2, aún sin evaluar. Se procede entonces a la activación/demanda de la ligadura de x_2:

HEAD-*stream* COMMUNICATION DEMAND



Es evidente que en el sistema bajo consideración ha de existir al menos otro proceso que no se ha detallado, en el que en particular aparecerán también ch_6 y ch_7.

□

En total, las reglas que regulan los procesos de desbloqueo, desactivación y demandas globales del subconjunto Jauja básico con *streams* incluye las reglas: WHNF UNBLOCKING, WHNF DEACTIVATION, BLOCKING PROCESS CREATION, PROCESS CREATION DEMAND, VALUE COMMUNICATION DEMAND, y HEAD-*stream* COMMUNICATION DEMAND:

$$\overset{Unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow} ; \overset{hsComd}{\Longrightarrow} .$$

Los pasos a seguir para demostrar que la sucesión de reglas expuestas es adecuada son la comprobación de la no interferencia de una regla consigo misma y de cada regla con las que le siguen.

Las auto-interferencias de todas salvo una de las reglas globales fueron ya estudiadas en la Sección 6.2.2.3. Resta por analizar la relativa a $\overset{hsComd}{\Longrightarrow}$.

Proposición 6.16 *Sea S un sistema, H_1 un heap de S, $x_1 \overset{I}{\mapsto} W_1$ una ligadura de H_1 y $x_1' \overset{I}{\mapsto} E_1 \in \mathsf{nf}(W_1, H_1)$. Sea H_2 un segundo heap de S, posiblemente el propio H_1, $x_2 \overset{I}{\mapsto} W_2$ una ligadura de H_2 con $x_2' \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$, con $x_1' \neq x_2'$. Entonces la aplicación de la regla $\overset{hsComd}{\longrightarrow}$ sobre x_1' no impide que posteriormente se pueda aplicar esta misma regla sobre x_2'.*

Demostración P.6.16
En la Sección A.9.

La no existencia de auto-interferencias, como puede observarse, es importante en tanto que las variables a activar sean distintas. Si nos encontramos en el otro caso posible, i.e. $x_1' = x_2'$, y aplicamos la regla HEAD-*stream* COMMUNICATION DEMAND sobre x_1 y x_1', el único cambio reside en la activación de la ligadura de x_1'; pero aunque esta aplicación deshabilita la posible aplicación de HEAD-*stream* COMMUNICATION DEMAND sobre x_2 y x_2', en el sistema ya se ha producido el mismo efecto que se hubiera producido con la aplicación de dicha regla, es decir, la ligadura de x_2' se ha activado.

Tratemos ahora la no-interferencia del resto de reglas con $\overset{hsComd}{\Longrightarrow}$.

Proposición 6.17 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 un segundo heap de S, posiblemente el propio H_1, $x_2 \overset{I}{\mapsto} W_2$ una ligadura-cabeza-inactiva de H_2 con $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$. Entonces la aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{hsComd}{\longrightarrow}$ sobre x_2.*

Demostración P.6.17
En la Sección A.9.

Proposición 6.18 *Sea S un sistema, H_1 un heap de S y $\theta_1 \overset{A}{\mapsto} W_1$ una ligadura de H_1. Sea H_2 un segundo heap de S, posiblemente el propio H_1, $x_2 \overset{I}{\mapsto} W_2$ una ligadura-cabeza-inactiva de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$. Entonces la aplicación de la regla $\overset{deact}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{hsComd}{\longrightarrow}$ sobre x_2.*

Demostración P.6.18
En la Sección A.9.

El único cambio que realiza $\overset{deact}{\longrightarrow}$ es la desactivación de las ligaduras asociadas con valores *whnf*. Dichas ligaduras no pueden encontrarse en $\mathsf{nf}(y_2, H_2)$, por lo que la desactivación no conduce a una activación por demanda posterior, es decir, la desactivación de ligaduras no habilita nuevas demandas por creación de procesos.

Las consideraciones que se hicieron en la Sección 6.2.2.3 con respecto a la relación entre $\overset{bpc}{\longrightarrow}$ y las reglas de demanda siguen siendo válidas en este punto. En cuanto al orden entre las demandas de creación de proceso, $\overset{pcd}{\longrightarrow}$, $\overset{vComd}{\longrightarrow}$ y $\overset{hsComd}{\longrightarrow}$, se tiene que el mismo es irrelevante dado que las ligaduras a activar están determinadas. De modo que la única interferencia posible es que una ligadura pudiera ser demandada inicialmente por más de una razón. Sin embargo, en cuanto que la primera de las reglas la demandara el resto ya encontraría realizada esta tarea.

Como en el caso de Jauja básico, demostraremos que la nueva transición global $\overset{Unbl}{\Longrightarrow}$ está bien definida. La certeza sobre esto está asegurada si cada una de las transiciones globales que la componen está bien definida. La de $\overset{wUnbl}{\Longrightarrow}$, $\overset{deact}{\Longrightarrow}$, $\overset{bpc}{\Longrightarrow}$ y $\overset{pcd}{\Longrightarrow}$, que no han cambiado, se mantienen, como quedó indicado por las Proposiciones A.13, A.14, A.15 y A.16, respectivamente. Que $\overset{vComd}{\Longrightarrow}$ esté bien definida está garantizado, la regla no ha cambiado con respecto a la de la Figura 6.6, salvo en el requerimiento de que W no sea lista. Sin embargo, en aquella ocasión las listas no existían, por lo que esta restricción se hallaba implícita.

Nos queda entonces por ver que $\overset{hsComd}{\Longrightarrow}$ está bien definida.

Proposición 6.19 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{hsCom}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{hsCom}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.19
En la Sección A.10.

Con esta demostración se asegura que $\overset{Unbl}{\Longrightarrow}$ está bien definida:

Corolario 6.1 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{Unbl}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{Unbl}{\Longrightarrow}$ para S está bien definida.*

Demostración C.6.1
Por las Proposiciones A.13, A.14, A.15, A.16 A.17 y 6.19 cada paso reiterado de los que componen $\overset{Unbl}{\Longrightarrow}$ está bien definido, por lo que la composición secuencial de ellos también. En consecuencia, la imagen de S por $\overset{Unbl}{\Longrightarrow}$ está bien definida.

c.q.d.

Y ahora sólo nos queda por ver que la regla $\overset{hsComd}{\Longrightarrow}$ transforma sistemas finitos en sistemas finitos. En cuanto a la regla $\overset{vComd}{\Longrightarrow}$, ello se verifica, como vimos en la Sección 6.2.2.3.

Proposición 6.20 *Sea S un sistema finito al que se le aplica $\overset{hsComd}{\Longrightarrow}$, $S \overset{hsComd}{\Longrightarrow} S'$, entonces el sistema S' es también finito.*

Demostración P.6.20
En efecto, la regla global $\overset{hsComd}{\longrightarrow}$ no modifica el número de ligaduras del sistema origen, tan solo transforma algunas de sus etiquetas. Por ello, independientemente del número de pasos simples incluidos en $\overset{hsComd}{\Longrightarrow}$, el cardinal de ligaduras de S no varía y, por lo tanto, S' es también finito.

c.q.d.

La ampliación de las Proposiciones 6.7 y 6.8 se realizará al finalizar las incorporaciones en la Sección 6.5.

Queda entonces demostrada la adecuación de la inclusión de las nuevas reglas para *streams*.

6.4. No-determinismo explícito

Consideramos a continuación la introducción de no-determinismo explícito, mediante el operador $\bowtie$, cuya sintaxis restringida es:

$$E \quad ::= \quad x_1 \bowtie x_2$$

Veamos las nuevas transiciones, tanto locales como globales, que precisa esta nueva incorporación.

6.4.1. Transiciones locales

Las transiciones locales necesarias para la evaluación de expresiones de mezcla deben llevar a cabo tres tareas:

- Incorporar elementos nuevos a la lista resultado.
- Cerrar la lista resultado cuando ambos *streams* argumento sean vacíos.
- Bloquear la mezcla en los casos en los que no sea posible incorporar un elemento a la lista, pero exista la posibilidad de incorporar nuevos elementos en el futuro.

6.4.1.1. Incorporación de elementos

El operador $\bowtie$ construye una lista cuyos elementos se encuentran todos en forma normal. En [Hen82] Henderson resalta que para que este tipo de operador sea una función la evaluación debería ser guiada por la demanda, en la que cuando no se dispusiera de un valor para incorporar a la mezcla se demandaría la evaluación de uno de los operandos, pero esta demanda siempre se ejercería sobre el mismo argumento. Sin embargo, este comportamiento otorgaría al operador $\bowtie$ propiedades no deseadas, sería una versión demasiado determinista.

Teniendo en cuenta estas observaciones, para nosotros en la expresión $x_1 \bowtie x_2$, no existirá una demanda explícita sobre ninguna de las dos variables en concreto. Nosotros en el caso de la semántica mínima haremos que prevalezca el deseo de terminar la evaluación de la variable *main* sobre cualquier otra propiedad, por lo que si la evaluación de la mezcla es necesaria para la evaluación de la variable *main*, sí que procederemos a incluir ambas variables, o las que de ellas se deriven, en la precedencia de la variable principal. En cualquier caso, la intención inicial es que cada x_i ($i \in \{1, 2\}$) esté ligada a la salida en forma de *stream* por un canal. Cada *stream* será evaluado de manera impaciente por sendos procesos. En el momento en que uno de los dos argumentos de esta mezcla tenga una forma normal como cabeza, este valor se convertirá en parte de la lista resultado. El no-determinismo aparece cuando ambos argumentos tienen una forma normal en su cabeza, manteniéndose esta cantidad de no-determinismo al no dar preferencia a ninguno de los argumentos. Las reglas que rigen este comportamiento se recogen en la Figura 6.12.

(left merge)

$$H' = H + \{x_1 \overset{I}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}$$
$$\text{si } \mathsf{nf}(W, H') = \emptyset$$
$$\mathit{fresh}(z)$$
$$H + \{x_1 \overset{I}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{I}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} [x_1^1 : z], z \overset{A}{\mapsto} x_1^2 \bowtie x_2\}$$

(right merge)

$$H' = H + \{x_2 \overset{I}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}$$
$$\text{si } \mathsf{nf}(W, H') = \emptyset$$
$$\mathit{fresh}(z)$$
$$H + \{x_2 \overset{I}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_2 \overset{A}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} [x_2^1 : z], z \overset{A}{\mapsto} x_1 \bowtie x_2^2\}$$

Figura 6.12: Jauja básico con *streams* y no-determinismo: reglas locales de mezcla

Si es la variable de la izquierda de $\bowtie$, x_1, la que se encuentra ligada a una lista, $[x_1^1 : x_1^2]$, y la cabeza de ésta, x_1^1 está ligada ligada a una "forma normal", la regla que se aplica es LEFT MERGE. La variable antes ligada a la expresión de mezcla queda ahora ligada a una lista, cuya cabeza es la variable x_1^1, y cuya cola, z, queda asociada a una nueva expresión de mezcla. Esta nueva expresión tiene por lado derecho la misma variable que la expresión inicial. Sin embargo, el lado izquierdo es ahora la cola de la lista/*stream* $[x_1^1 : x_1^2]$, i.e. x_1^2.

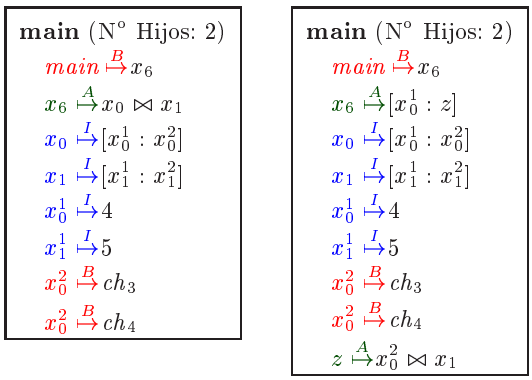
La regla RIGHT MERGE es simétrica a la anterior, y se aplica cuando es la variable de la derecha la que verifica la condición solicitada.

Cuando las variables de ambos lados se encuentran ligadas a sendas listas con cabezas en "forma normal", se elegirá de manera no-determinista la regla (solamente una) a aplicar.

Ejemplo 6.29 *Mezcla no-determinista.*

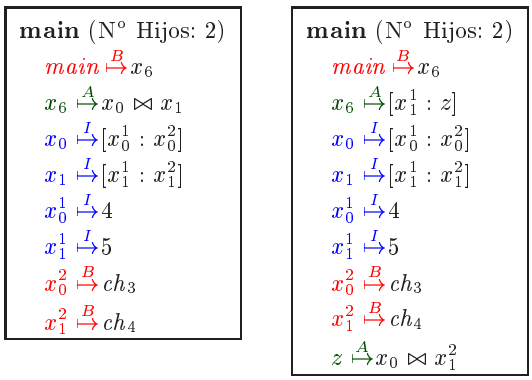
Veamos parte de la evaluación del Ejemplo 4.4, cuando estamos en disposición de realizar la mezcla, pudiendo colaborar en ella cualquiera de los argumentos. Observemos primero el caso de la aplicación de la mezcla por la izquierda:

LEFT MERGE



En este caso, el valor 4 ha pasado a formar parte de la mezcla. Mas exactamente, ello sólo es así indirectamente, puesto que realmente la cabeza no es 4, sino la variable que se encuentra ligada a 4. Veamos la alternativa contraria, en la que es el valor 5 el que pasa a ser cabeza de la mezcla:

RIGHT MERGE



Como quiera que ambas reglas son aplicables, estamos ante un caso de no-determinismo. Entonces es posible que el cómputo siga tras incorporar como cabeza de la mezcla la cabeza de cualquiera de las dos listas argumento.

□

Ambas reglas de mezcla únicamente introducen una ligadura nueva en el sistema, por lo que si el sistema inicial es finito el transformado también es finito.

6.4.1.2. Cierre de mezcla

No obstante, con mezclar no basta, hay que determinar cuándo se termina de realizar esta mezcla, y solamente hay un caso en el que se da por cerrada esta operación: cuando ambas listas/*streams* están vacías. Esta finalización la lleva a cabo la regla MERGE TERMINATION de la Figura 6.13.

(merge termination)

$$H + \{x_1 \overset{I}{\mapsto} \texttt{nil}, x_2 \overset{I}{\mapsto} \texttt{nil}\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow H + \{x_1 \overset{I}{\mapsto} \texttt{nil}, x_2 \overset{I}{\mapsto} \texttt{nil}, \theta \overset{A}{\mapsto} \texttt{nil}\}$$

Figura 6.13: Jauja básico con *streams* y no-determinismo: regla local fin de mezcla

Se observa que las variables afectadas por $\bowtie$ están asociadas al valor *whnf* `nil`, de lo que deducimos que la mezcla no va a poder incrementarse con más elementos, por lo que la variable de la mezcla se liga también al valor `nil`.

Nótese también que el número de ligaduras del sistema transformado es el mismo que el del sistema inicial. En consecuencia, si el segundo es finito el primero también lo es.

6.4.1.3. Bloqueo de mezcla

El último grupo de transiciones locales para $\bowtie$ se encarga de bloquear la mezcla cuando:

1. Sus argumentos no son *streams*, o si uno de ellos es el *stream* vacío y el otro aún no se ha decantado sobre si es o no una lista.

2. Uno de sus argumentos no es lista o es la lista vacía, y el otro sí que lo es, pero su cabeza no es un valor comunicable.

3. Ambos argumentos son listas no vacías, pero ninguna de las cabezas es un valor comunicable.

Obsérvese que solamente se produce el bloqueo de la mezcla, pero la demanda no se realiza (recordemos los argumentos de [Hen82]). Las reglas que formalizan estas restricciones quedan recogidas en la Figura 6.14.

Expliquemos cada una de estas reglas de bloqueo con una serie de ejemplos.

Ejemplo 6.30 *Bloqueo de mezcla: caso 1.*

En el presente ejemplo la parte izquierda de la mezcla se encuentra ligada a una expresión que no es un valor *whnf*, por ende tampoco una lista, en tanto que la parte derecha está asociada a la lista vacía. Bajo estas condiciones la mezcla no puede continuar, ni tampoco ser cerrada, por lo que se procede a su bloqueo aplicando la regla BLOCKING MERGE 1:

(blocking merge 1)

$$\text{si } E_1, E_2 \notin \mathit{List} \vee (E_1 \notin \mathit{List} \wedge E_2 \equiv \texttt{nil}) \vee (E_2 \notin \mathit{List} \wedge E_1 \equiv \texttt{nil})$$
$$H + \{x_1 \overset{\alpha_1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow H + \{x_1 \overset{\alpha_1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

(blocking merge 2)

$$H' = H + \{x_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{\alpha_1^1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} E_2, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}$$
$$\text{si } (\mathtt{nf}(x_1^1, H') \neq \emptyset) \wedge (E_2 \notin \mathit{List} \vee E_2 \equiv \texttt{nil})$$
$$H + \{x_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{\alpha_1^1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{\alpha_1^1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

(blocking merge 3)

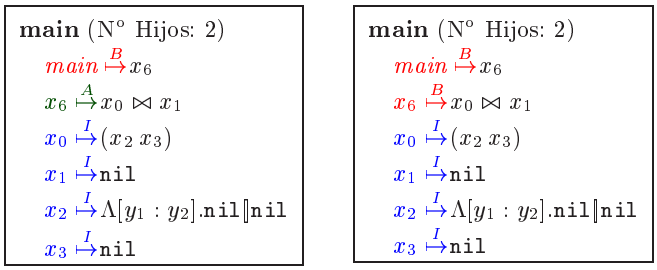
$$H' = H + \{x_1 \overset{\alpha_1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{\alpha_2^1}{\mapsto} E_2, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}$$
$$\text{si } (E_1 \notin \mathit{List} \vee E_1 \equiv \texttt{nil}) \wedge (\mathtt{nf}(x_2^1, H') \neq \emptyset)$$
$$H + \{x_1 \overset{\alpha_1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{\alpha_2^1}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha_1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{\alpha_2^1}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

(blocking merge 4)

$$H' = H + \{x_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{\alpha_1^1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{\alpha_2^1}{\mapsto} E_2, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}$$
$$\text{si } [(E_1 \notin \mathit{Val} \vee (E_1 \in \mathit{Val} \wedge \mathtt{nf}(E_1, H') \neq \emptyset)) \wedge (E_2 \notin \mathit{Val} \vee (E_2 \in \mathit{Val} \wedge \mathtt{nf}(E_2, H') \neq \emptyset))]$$
$$H + \{x_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{\alpha_1^1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{\alpha_2^1}{\mapsto} E_2\} : \theta \overset{A}{\mapsto} x_1 \bowtie x_2 \longrightarrow$$
$$\longrightarrow H + \{x_1 \overset{\alpha_1}{\mapsto} [x_1^1 : x_1^2], x_1^1 \overset{\alpha_1^1}{\mapsto} E_1, x_2 \overset{\alpha_2}{\mapsto} [x_2^1 : x_2^2], x_2^1 \overset{\alpha_2^1}{\mapsto} E_2, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}$$

Figura 6.14: Jauja básico con *streams* y no-determinismo: reglas locales de bloqueo de mezcla

BLOCKING MERGE 1



Esta regla también se aplicaría si fuera x_1 la que estuviera ligada a una expresión que no se encontrara en *whnf* ni fuera lista, independientemente de si x_0 se encontrara ligada a la lista vacía o a una expresión que no fuera lista.

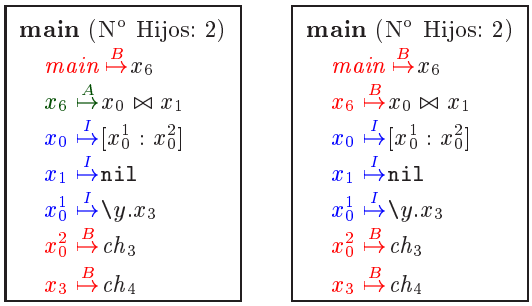
□

Ejemplo 6.31 *Bloqueo de mezcla: caso 2.*

En este ejemplo una de las partes de la mezcla es la lista vacía, y la otra es una lista

no vacía. La cabeza de dicha lista sí que es un *whnf*, pero no tiene todas sus dependencias libres resueltas, por lo que aún no puede formar parte de la mezcla.

BLOCKING MERGE 2



Obsérvese que un caso como el descrito solamente es posible si es el propio proceso el que ha evaluado la variable x_0^1. El desbloqueo se producirá en los siguientes casos:

- Provocado por la evaluación de x_1, variable derecha de la mezcla; o bien
- la variable x_3 es evaluada porque otra variable ajena a la mezcla la demanda; o bien
- la variable x_3 es receptora de canal.

En el presente caso solamente es posible el tercer caso.

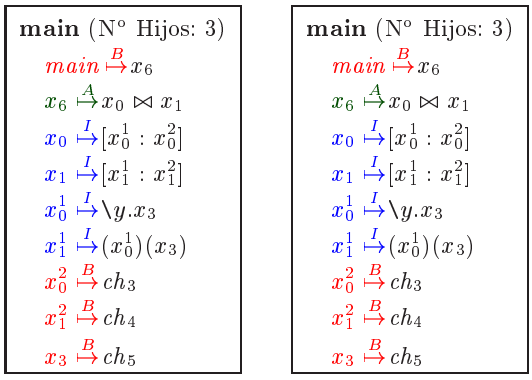
□

La regla BLOCKING MERGE 3 es simétrica a la regla BLOCKING MERGE 2, intercambiando las variables izquierda y derecha de la mezcla.

Ejemplo 6.32 *Bloqueo de mezcla: caso 4.*

En este caso ambas partes de la mezcla se encuentran ligadas a listas no vacías. Sin embargo, ninguna de sus cabezas está preparada para formar parte de la mezcla: una parte no puede contribuir porque está pendiente de resolver la dependencia libre de x_3, mientras que la otra no contribuye por no estar ligada a un valor *whnf*. En consecuencia se procede al bloqueo:

BLOCKING MERGE 4



□

En todas las reglas de bloqueo de mezcla de la Figura 6.14 se mantiene intacto el número de ligaduras del sistema, por lo que transforman sistemas finitos en sistemas también finitos.

El resto de tareas relativas al no-determinismo explícito se enmarcan en el nivel global de transiciones.

6.4.2. Transiciones globales

Si antes hemos procedido a bloquear mezclas, en algún momento tendrá que tener lugar el desbloqueo de éstas. La condición para ello es que al menos uno de sus argumentos sea un *stream* con cabeza comunicable, o que ambos sean *streams* finalizados. Estos desbloqueos son realizados aplicando las reglas de la Figura 6.15, donde $E^x_{\bowtie}$ equivale a una expresión bloqueada en una mezcla esperando x, es decir, $x \bowtie y$ o $y \bowtie x$.

(unblocking merge 1)

$$H' = H + \{x \overset{A}{\mapsto} [x_1 : x_2], x_1 \overset{I}{\mapsto} W, \theta \overset{B}{\mapsto} E^x_{\bowtie}\}$$
$$\text{si } (\mathsf{nf}(W, H') = \emptyset)$$
$$(S, \langle p, H + \{x \overset{AI}{\mapsto} [x_1 : x_2], x_1 \overset{AI}{\mapsto} W, \theta \overset{B}{\mapsto} E^x_{\bowtie}\}\rangle) \overset{mUnbl1}{\longrightarrow} (S, \langle p, H + \{x \overset{AI}{\mapsto} [x_1 : x_2], x_1 \overset{AI}{\mapsto} W, \theta \overset{A}{\mapsto} E^x_{\bowtie}\}\rangle)$$

(unblocking merge 2)

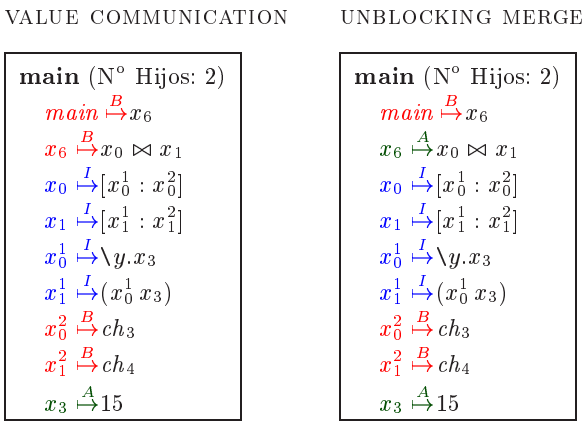
$$(S, \langle p, H + \{x_1 \overset{AI}{\mapsto} \mathtt{nil}, x_2 \overset{AI}{\mapsto} \mathtt{nil}, \theta \overset{B}{\mapsto} x_1 \bowtie x_2\}\rangle) \overset{mUnbl2}{\longrightarrow} (S, \langle p, H + \{x_1 \overset{AI}{\mapsto} \mathtt{nil}, x_2 \overset{AI}{\mapsto} \mathtt{nil}, \theta \overset{A}{\mapsto} x_1 \bowtie x_2\}\rangle)$$

Figura 6.15: Jauja básico con *streams* y no-determinismo: reglas globales de desbloqueo

La regla UNBLOCKING MERGE 1 se aplica en el primer caso, es decir, cuando una de las cabezas se ha evaluado ya a forma normal, mientras que la regla UNBLOCKING MERGE 2 se aplica en el segundo caso, es decir, cuando ambas listas de la mezcla ya son vacías.

Ejemplo 6.33 *Desbloqueo de mezcla: cabeza preparada.*

En este caso una de las partes tiene su cabeza preparada para formar parte de la mezcla, por lo que se procede al desbloqueo:



□

Estas nuevas reglas de desbloqueo se han de intercalar entre las anteriores, dando lugar a la siguiente regla de paso reiterado:

$$\overset{Unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow} ; \overset{mUnbl1}{\Longrightarrow} ; \overset{mUnbl2}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow} ; \overset{hsComd}{\Longrightarrow} .$$

La ausencia de interferencias entre las dos nuevas reglas globales incorporadas y las que ya componían $\overset{Unbl}{\Longrightarrow}$ se demuestra de manera análoga a como ya se hizo con $\overset{wUnbl}{\longrightarrow}$ respecto de las demás en la Sección A.3, siendo el principal argumento que la única ligadura modificada por una regla de desbloqueo es la que antes estaba bloqueada y ahora pasa a ser activa, no modificándose así ninguna de las ligaduras a las que se puede aplicar cualquiera de las transiciones globales que le siguen en $\overset{Unbl}{\Longrightarrow}$.

Como ya se demostró que $\overset{Unbl}{\Longrightarrow}$ estaba bien definida (Sección 6.2.2.3), sólo hay que demostrar que tanto $\overset{mUnbl1}{\Longrightarrow}$ como $\overset{mUnbl2}{\Longrightarrow}$ también están bien definidas cuando se parte de un sistema finito.

Tratemos primero el paso $\overset{mUnbl1}{\Longrightarrow}$.

Proposición 6.21 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{mUnbl1}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{mUnbl1}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.21
En la Sección A.11.

Consideremos ahora el paso global $\overset{mUnbl2}{\Longrightarrow}$.

Proposición 6.22 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{mUnbl2}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{mUnbl2}{\Longrightarrow}$ para S está bien definida..*

Demostración P.6.22
En la Sección A.12.

Es evidente que ambas reglas globales de desbloqueo transforman un sistema finito en otro que también lo es, pues no alteran el número de ligaduras del sistema.

Del mismo modo que sucedió en el caso de la extensión con *streams*, la función que devuelve las expresiones dependientes ha de ser extendida por la incorporación del no-determinismo.

$$\texttt{dexp}(E_1 \bowtie E_2) = \{E_1, E_2\}$$

Figura 6.16: Jauja básico con *streams* y no-determinismo: detección de dependencias

El resto de funciones no se ven alteradas ya que las definiciones nh y nf de la Figura 6.4 ya recogen el caso de variables ligadas a una mezcla.

Una vez demostrado que $\overset{Unbl}{\Longrightarrow}$ está bien definida y extendidas las funciones necesarias, se da por finalizada la dotación de semántica a la extensión de Jauja con no-determinismo.

6.5. Canales dinámicos

Recordemos que la sintaxis restringida correspondiente a las estructuras sintácticas que se utilizaban para la creación y uso de canales dinámicos son:

E	$::=$	$\texttt{new}(y, x)E$	canales dinámicos
	$\mid$	$x\,!\,y_1$ `par` y_2	conexión dinámica

Un canal dinámico se crea en dos fases:

1. El proceso que recibirá datos a través de dicho canal comunica el nombre que asocia en su *heap* a este canal.
2. Uno —y sólo uno— de los procesos a los que este nombre ha llegado conecta con el proceso inicial.

De estas tareas, la primera, así como la demanda para obtener el nombre del canal por el que se va a comunicar, se lleva a cabo mediante transiciones locales. Sin embargo, la conexión del canal corresponde a una regla global.

6.5.1. Transiciones locales

Las transiciones locales para la creación de canales dinámicos se muestran en la Figura 6.17.

$$H : \theta \overset{A}{\mapsto} \texttt{new}(y, x)\ \texttt{in}\ E \longrightarrow H + \{\theta \overset{A}{\mapsto} E[y'/y, x'/x], y' \overset{I}{\mapsto} \mathsf{ch}(self, x'), x' \overset{B}{\mapsto} \lightning\} \quad \textbf{(channel creation)}$$

$$\text{si } E \notin Val \qquad \textbf{(channel name demand)}$$
$$H + \{z \overset{IAB}{\mapsto} E\} : \theta \overset{A}{\mapsto} z\,!\,y_1\ \texttt{par}\ y_2 \longrightarrow H + \{z \overset{AAB}{\mapsto} E, \theta \overset{B}{\mapsto} z\,!\,y_1\ \texttt{par}\ y_2\}$$

Figura 6.17: Jauja: reglas locales para canales dinámicos

Cuando se evalúa la expresión $\texttt{new}(y, x)E$ se crea un canal nuevo. En esta expresión se declara una variable local, x, cuyo ámbito es la expresión E; será en E donde se consuma el valor comunicado a través de este nuevo canal. La variable x se liga a la salida del canal, cuyo productor se determinará más tarde. Como en un primer momento no se ha establecido la conexión, x queda inicialmente ligada al valor especial $\lightning$ (desconectado), y la correspondiente ligadura se bloquea. Por otra parte, la variable y —que representa el nombre del canal— queda ligada a otro valor especial: $\mathsf{ch}(self, x)$.[4] Este valor es el que será enviado a través de otros canales, hasta que un proceso decida conectar y convertirse en el productor. Todas estas tareas son realizadas por la regla CHANNEL CREATION.

La otra expresión relacionada con la gestión de canales dinámicos es $z\,!\,y_1\ \texttt{par}\ y_2$. Su evaluación corresponde a las transiciones globales, pero para poder conectar el canal es necesario que la variable z se encuentre ligada al nombre del canal, por lo que se inicia la demanda correspondiente CHANNEL NAME DEMAND y se produce un bloqueo hasta que se obtenga el nombre del canal.

En las dos reglas de la Figura 6.17, partiendo de un sistema finito se obtiene otro que también lo es. Ello es debido a que la regla CHANNEL CREATION únicamente aumenta en dos el número de ligaduras del sistema inicial, mientras que la regla CHANNEL NAME DEMAND no modifica dicho número.

6.5.2. Transiciones globales

Un proceso p que ha recibido el nombre de un canal dinámico $\mathsf{ch}(c, x)$ —ligado a una variable z— se convierte en productor evaluando la expresión $z\,!\,y_1\ \texttt{par}\ y_2$. La transición global de conexión se muestra en la Figura 6.18.

Se crea una nueva salida, o, que se liga a la variable y_1. Por otra parte, la variable x, donde se van a recibir los valores comunicados, se queda bloqueada en esta nueva salida. Mientras, la evaluación de la expresión completa continúa con y_2. Se observa que,

[4] *self* denota el nombre del proceso sobre el que se aplica la regla de creación, es decir, el propietario del *heap*.

(channel connection)

$$(S, \langle p, H_p + \{z \overset{I}{\mapsto} \mathsf{ch}(c,x), \theta \overset{AB}{\mapsto} z\,!\,y_1\ \mathtt{par}\ y_2\}\rangle, \langle c, H_c + \{x \overset{B}{\mapsto} \lightning, y \overset{I}{\mapsto} \mathsf{ch}(c,x)\}\rangle) \overset{conn}{\longrightarrow}$$
$$\overset{conn}{\longrightarrow} (S, \langle p, H_p + \{\theta \overset{A}{\mapsto} y_2, o \overset{A}{\mapsto} y_1, z \overset{I}{\mapsto} \mathsf{ch}(c,x)\}\rangle, \langle c, H_c + \{x \overset{B}{\mapsto} o, y \overset{I}{\mapsto} \mathsf{ch}(c,x)\}\rangle)$$

Figura 6.18: Jauja: regla global de conexión de canales dinámicos

inicialmente, x ha de estar bloqueada en $\lightning$, es decir, solamente es posible la conexión con un canal desconectado. En este momento se tienen que realizar todas las conexiones posibles, para lo cual se recurre de nuevo a una transición de paso reiterado: $\overset{conn}{\Longrightarrow}$.

A diferencia de otras transiciones globales de paso reiterado, esta no queda enmarcada en $\overset{Unbl}{\Longrightarrow}$, sino que es una de las partes de la evolución del sistema:

$$\overset{sys}{\Longrightarrow} = \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{Unbl}{\Longrightarrow} ; \overset{conn}{\Longrightarrow}$$

Que $\overset{sys}{\Longrightarrow}$ estará bien definida si cada una de las transiciones que la componen lo están. Como para las tres primeras ya lo demostramos, falta demostrar que la regla global CHANNEL CONNECTION está bien definida:

Proposición 6.23 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{conn}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{conn}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.23
En la Sección A.13.

Al igual que sucedía con otras reglas globales, CHANNEL CONNECTION transforma un sistema finito en otro que también lo es, ya que solamente incrementa en uno el número de ligaduras.

Ejemplo 6.34 *Generación y uso de un canal dinámico.*

$$\mathtt{new}(y,x).\mathtt{let}\ x_1 = \backslash z.z\,!\,x_2\ \mathtt{par}\ x_3, x_2 = 1, x_3 = 2\ \mathtt{in}\ x_1\#y$$

Suponiendo el marco de la semántica mínima, la evaluación de esta expresión procede de la siguiente forma:

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \mathtt{new}(y,x).\mathtt{let}\ x_1 = \backslash z.z\,!\,x_2\ \mathtt{par}\ x_3, x_2 = 1, x_3 = 2, x_4 = x_1\#y_0\ \mathtt{in}\ x_4$

Se crea la entrada a *main* del nuevo canal dinámico.

main (N° Hijos: 0)
$main \overset{A}{\mapsto} \texttt{let } x_1 = \backslash z.z \,!\, x_2 \texttt{ par } x_3, x_2 = 1, x_3 = 2, x_4 = x_1 \texttt{\#} y_0 \texttt{ in } x_4$
$y_0 \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$
$x_0 \overset{B}{\mapsto} \lightning$

Se introducen las variables locales en el *heap* y se crea el nuevo proceso.

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_8$	$x_{11} \overset{I}{\mapsto} \backslash x_{15}.x_{15} \,!\, x_{13} \texttt{ par } x_{14}$
$y_0 \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$	$x_{12} \overset{B}{\mapsto} ch_{10}$
$x_0 \overset{B}{\mapsto} \lightning$	$x_{13} \overset{I}{\mapsto} 1$
$x_5 \overset{I}{\mapsto} \backslash z.z \,!\, x_6 \texttt{ par } x_7$	$x_{14} \overset{I}{\mapsto} 2$
$x_6 \overset{I}{\mapsto} 1$	$ch_9 \overset{A}{\mapsto} (x_{11})(x_{12})$
$x_7 \overset{I}{\mapsto} 2$	
$x_8 \overset{B}{\mapsto} ch_9$	
$ch_{10} \overset{A}{\mapsto} y_0$	

Se desarrolla la aplicación del canal ch_9, llegándose al punto de demanda, para poder enviar por el canal dinámico.

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_8$	$x_{11} \overset{I}{\mapsto} \backslash x_{15}.x_{15} \,!\, x_{13} \texttt{ par } x_{14}$
$y_0 \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$	$x_{12} \overset{B}{\mapsto} ch_{10}$
$x_0 \overset{B}{\mapsto} \lightning$	$x_{13} \overset{I}{\mapsto} 1$
$x_5 \overset{I}{\mapsto} \backslash z.z \,!\, x_6 \texttt{ par } x_7$	$x_{14} \overset{I}{\mapsto} 2$
$x_6 \overset{I}{\mapsto} 1$	$ch_9 \overset{B}{\mapsto} x_{12} \,!\, x_{13} \texttt{ par } x_{14}$
$x_7 \overset{I}{\mapsto} 2$	
$x_8 \overset{B}{\mapsto} ch_9$	
$ch_{10} \overset{A}{\mapsto} y_0$	

Tras la aplicación de VALUE con las variables ch_{10} e y_0, se comunica el nombre del canal dinámico. Posteriormente se desactiva la variable receptora.

main (N° Hijos: 1)	**p₁** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_8$	$x_{11} \overset{I}{\mapsto} \backslash x_{15}.x_{15} \,!\, x_{13} \texttt{ par } x_{14}$
$y_0 \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$	$x_{12} \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$
$x_0 \overset{B}{\mapsto} \lightning$	$x_{13} \overset{I}{\mapsto} 1$
$x_5 \overset{I}{\mapsto} \backslash z.z \,!\, x_6 \texttt{ par } x_7$	$x_{14} \overset{I}{\mapsto} 2$
$x_6 \overset{I}{\mapsto} 1$	$ch_9 \overset{B}{\mapsto} x_{12} \,!\, x_{13} \texttt{ par } x_{14}$
$x_7 \overset{I}{\mapsto} 2$	
$x_8 \overset{B}{\mapsto} ch_9$	

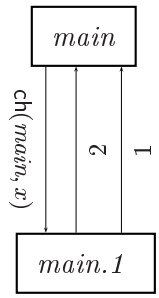
El sistema se encuentra en condiciones de conectar los procesos main y p_1 con el nuevo canal.

main (N° Hijos: 1)	**p$_1$** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_8$	$x_{11} \overset{I}{\mapsto} \backslash x_{15}.x_{15} \,!\, x_{13}$ par x_{14}
$y_0 \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$	$x_{12} \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$
$x_0 \overset{B}{\mapsto} ch_{15}$	$x_{13} \overset{I}{\mapsto} 1$
$x_5 \overset{I}{\mapsto} \backslash z.z \,!\, x_6$ par x_7	$x_{14} \overset{I}{\mapsto} 2$
$x_6 \overset{I}{\mapsto} 1$	$ch_9 \overset{A}{\mapsto} x_{14}$
$x_7 \overset{I}{\mapsto} 2$	$ch_{15} \overset{A}{\mapsto} x_{13}$
$x_8 \overset{B}{\mapsto} ch_9$	

Los restantes pasos dentro de este marco mínimo serían la aplicación de VALUE, la comunicación del valor 2 y la desactivación de la variable *main*.

main (N° Hijos: 1)	**p$_1$** (N° Hijos: 0)
$main \overset{I}{\mapsto} 2$	$x_{11} \overset{I}{\mapsto} \backslash x_{15}.x_{15} \,!\, x_{13}$ par x_{14}
$y_0 \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$	$x_{12} \overset{I}{\mapsto} \mathsf{ch}(main, x_0)$
$x_0 \overset{B}{\mapsto} ch_{15}$	$x_{13} \overset{I}{\mapsto} 1$
$x_5 \overset{I}{\mapsto} \backslash z.z \,!\, x_6$ par x_7	$x_{14} \overset{I}{\mapsto} 2$
$x_6 \overset{I}{\mapsto} 1$	$ch_{15} \overset{A}{\mapsto} x_{13}$
$x_7 \overset{I}{\mapsto} 2$	
$x_8 \overset{I}{\mapsto} 2$	

En el caso de la semántica máxima sí que se produciría la comunicación a través del canal dinámico, creándose la siguiente topología:



En ella se observa como los dos canales habituales de envío de valores del padre al hijo y del hijo al padre se han visto aumentados por el nuevo canal dinámico.

□

Las funciones `dexp` y `nf` tienen que adaptarse a la incorporación de las estructuras sintácticas relativas a los canales dinámicos. Las correspondientes extensiones se recogen en la Figura 6.19.

La función `dexp` sobre $\mathsf{ch}(p, x)$ y $\lightning$ devuelve el conjunto vacío, pues son valores sin variables libres. En el caso de $z \,!\, E_1$ par E_2, el resultado lo constituyen E_1 y E_2,

$$\begin{aligned}\mathtt{dexp}(\mathsf{ch}(p,x)) &= \emptyset\\ \mathtt{dexp}(\lightning) &= \emptyset\\ \mathtt{dexp}(z\,!\,E_1\,\mathtt{par}\,E_2) &= \{E_1, E_2\}\\ \mathtt{dexp}(\mathtt{new}(y,x)\ \mathtt{in}\ E) &= \{E\}\end{aligned}$$

$$\mathsf{nh}(E,H) \models \text{(i1),(i2),(ii)} \wedge \forall A.(A \subseteq H \wedge A \models \text{(i1),(i2),(ii)} \Rightarrow \mathsf{nh}(E,H) \subseteq A)$$

$$\text{(i1)}\quad \nu \overset{\alpha}{\mapsto} E \in H \wedge E \not\equiv \mathsf{ch}(p,x) \wedge E \not\equiv \lightning \Rightarrow \{\nu \overset{I}{\mapsto} E\} \cup \mathsf{nh}(E,H) \subseteq \mathsf{nh}(\nu,H)$$

$$\text{(i2)}\quad \nu \overset{\alpha}{\mapsto} E \in H \wedge (E \equiv \mathsf{ch}(p,x) \vee E \equiv \lightning) \Rightarrow \emptyset \subseteq \mathsf{nh}(\nu,H)$$

$$\mathsf{nf}(E,H) \models \text{(i1),(i2)-(v)} \wedge \forall A.(A \subseteq H \wedge A \models \text{(i1),(i2)-(v)} \Rightarrow \mathsf{nf}(E,H) \subseteq A)$$

$$\text{(i1)}\quad x \overset{\alpha}{\mapsto} W \in H \wedge W \not\equiv \mathsf{ch}(p,x) \wedge W \not\equiv \lightning \Rightarrow \{x \overset{\alpha}{\mapsto} W\} \cup \mathsf{nf}(W,H) \subseteq \mathsf{nf}(W,H)$$

$$\text{(i2)}\quad x \overset{\alpha}{\mapsto} W \in H \wedge (W \equiv \mathsf{ch}(p,x) \vee W \equiv \lightning) \Rightarrow \emptyset \subseteq \mathsf{nf}(W,H)$$

Figura 6.19: Jauja: detección de dependencias para canales dinámicos

pues z tiene un carácter especial: cuando z se encuentre evaluada —suponiendo que estamos ante un programa correcto— tendrá que estar ligada a un valor de la forma $\mathsf{ch}(p,x)$. Se pretende que los valores de este tipo sean pasados de un proceso a otro vía comunicación, y no se desea que se realice la conexión de un canal dinámico al tratarse de un *efecto lateral* de copia de ligaduras de un *heap* a otro. Por ello, cuando se produzca la recolección de ligaduras, z no será considerada. Por otra parte, la demanda de evaluación de z es explícita con CHANNEL NAME DEMAND, y como no es una variable que se vaya a copiar, nf no debe recogerla como dependencia libre, aunque se encuentre sin evaluar. Estas consideraciones hacen necesaria la modificación tanto de la función nh, como de la función nf.

En cuanto al desbloqueo de ligaduras, entre las reglas locales necesarias para tratar con canales dinámicos se ha introducido una de demanda que bloquea la ligadura que necesita del nombre del canal. Por esta razón, extendemos E^x_B con $x\,!\,y_1\,\mathtt{par}\,y_2$. Se completa así la evaluación para crear canales dinámicos y transmitir valores a través de ellos.

Antes de estudiar propiedades de esta semántica falta por demostrar las adaptaciones de las Proposiciones 6.7 y 6.8, respectivamente:

Proposición 6.24 *Consideremos dos variables distintas, θ_1 y θ_2, y un heap que puede expresarse de dos modos distintos: $H = H^{(i,1)} + H^{(i,2)} + \{\theta_i \overset{A}{\mapsto} E_i\}$, con $i \in \{1,2\}$, de modo que en cada caso tengamos $H^{(i,1)} + H^{(i,2)} : \theta_i \overset{A}{\mapsto} E_i \longrightarrow H^{(i,1)} + K^{(i,2)}$. Entonces*

1. *Si $\theta' \overset{IB}{\mapsto} E' \in K^{(1,2)}$ tendremos $\theta' \notin dom(K^{(2,2)})$*
2. *Si $\theta' \overset{A}{\mapsto} E' \in K^{(1,2)}$ tendremos $\theta' \notin dom(K^{(2,2)})$ o bien $\theta' \overset{A}{\mapsto} E' \in K^{(2,2)}$*

Demostración P.6.24
En la Sección A.14.

Proposición 6.25 *Sean $E_0 \notin Val$ una expresión cerrada bien formada, y $\langle p_0, \{main \overset{A}{\mapsto} E_0\}\rangle \Longrightarrow^+ (S, \langle p, H + \{\theta \overset{A}{\mapsto} E\}\rangle)$ una secuencia de configuraciones con al menos un paso. Entonces, si E no es ni una* `#`*-expresión ni $z\,!\,y_1\,\mathtt{par}\,y_2$ tal que $z \overset{I}{\mapsto} \mathsf{ch}(c,x) \in H + \{\theta \overset{A}{\mapsto} E\}$, existirá H' tal que $H : \theta \overset{A}{\mapsto} E \longrightarrow H'$.*

Demostración P.6.25
En la Sección A.15.

6.6. Propiedades

La intención de esta sección es introducir elementos que nos permitan comparar programas desde el punto de vista de sus cómputos. Para ello daremos una noción particular de tiempo y definiremos diferentes índices del grado de paralelismo. Las medidas se definirán en la misma línea que se hizo para la semántica operacional de GPH en [BKHT99].

Es evidente que la semántica operacional definida dista aún mucho de ser una implementación realista de Eden. Sin embargo, dado el bajo nivel de abstracción empleado, siguiendo el cual se ha implementado un intérprete de Jauja básico, esta semántica sí nos permite extraer conclusiones sobre el grado de paralelismo y realizar comparaciones en relación al tiempo de ejecución.

Para las dos primeras definiciones vamos a considerar una secuencia de reducción finita, $S_0 \Longrightarrow S_1 \Longrightarrow \ldots \Longrightarrow S_n$, que correspondería a un programa cuya evaluación termina. De esa secuencia, para cada sistema S_i $(0 \leq i \leq n)$ y cada *heap* H^i_j $(1 \leq j \leq n_i,\ n_i = |S_i|)$ del sistema S_i, consideramos el conjunto $\mathcal{LD}(H^i_j)$, que era el conjunto de ligaduras activas del *heap* H^i_j que evolucionaban de manera local al aplicar la regla PARALLEL-P de la Figura 6.2.

Definición 6.10 *Tiempo y trabajo.*

- *Tiempo de ejecución:* $R = n$; es la noción particular de tiempo que mencionábamos.
- *Trabajo total realizado:* $T = \sum_{i=0}^{n-1} \sum_{j=1}^{n_i} |\mathcal{LD}(H^i_j)|$, es la cantidad de ligaduras activas que han evolucionado a lo largo del cómputo.

Estas definiciones valen tanto para la semántica mínima como para la máxima, e incluso para cualquier otra que se pudiera definir, pues en la propia definición de $\mathcal{LD}$ va incluido el grado de especulación.

□

Las siguientes medidas permiten cuantificar la cantidad de paralelismo que se ha explotado en el sistema.

Definición 6.11 *Paralelismo entre ligaduras.*

- *Paralelismo medio:* PM= $\frac{T}{R}$. Se contabilizan todas las ligaduras que han evolucionado y se reparten entre el tiempo del cómputo, obteniéndose la cantidad media de ligaduras activas que han evolucionado en un paso.
- *Paralelismo máximo:* PX= $\mathsf{max}\{\sum_{j=1}^{n_i} |\mathcal{LD}(H_j^i)|\}_{i=0}^{n-1}$. Esta medida es la cota superior del número de ligaduras desarrollables que han evolucionado en un paso de la secuencia.

□

En GpH solamente existía paralelismo entre las ligaduras del *heap*. Sin embargo, en Jauja el paralelismo se extiende también a los procesos. En consecuencia, también es de nuestro interés observar el grado de paralelismo entre procesos.

Definición 6.12 *Paralelismo entre procesos.*

- *Paralelismo medio de procesos:* PMp= $\frac{N}{R+1}$, donde $N = \sum_{i=0}^{n} n_i$ es la suma total de procesos que han existido en algún momento del cómputo.
- *Paralelismo máximo de procesos:* PXp= n_n.

El paralelismo medio toma el número total de procesos y los reparte entre el número de pasos, como si la evaluación hubiera contado con un número constante de procesos en todos los sistemas por los que ha ido evolucionando. En cuanto al paralelismo máximo, como un proceso cuando se crea nunca desaparece, el sistema final contiene todos los que se hayan creado a lo largo del cómputo.

□

Por último, introducimos una medida para discernir si el paralelismo introducido redunda en una mayor velocidad de ejecución. Para ello se comparan las secuencias de reducción correspondientes a la semántica mínima con las de la semántica máxima:

$$S_0^{min} \Longrightarrow S_1^{min} \Longrightarrow \ldots \Longrightarrow S_n^{min}$$
$$S_0^{max} \Longrightarrow S_1^{max} \Longrightarrow \ldots \Longrightarrow S_m^{max}$$

donde, evidentemente, $S_0^{min} = S_0^{max}$.

Definición 6.13 *Velocidad.*

V= $\frac{R_{min}}{R_{max}}$.

Se obtendrá una mayor velocidad en tanto menos sean los pasos necesarios para el caso de la semántica máxima.

□

Veamos la concreción de cada una de las medidas anteriores para los Ejemplos 6.20 y 6.21, correspondientes a las semánticas mínima y máxima, respectivamente.

Ejemplo 6.35 *Medidas de ejecución.*

	Mínima	Máxima
R	2	3
T	2	5
PM	1	$\frac{5}{3}\ (1,\hat{6})$
PX	1	3
PMp	$\frac{5}{3}\ (1,\hat{6})$	$\frac{7}{4}\ (1,75)$
PXp	2	2
V	$\frac{2}{3}\ (0,\hat{6})$	

A la vista de estas medidas podemos observar que, si bien en ambos casos se han creado la misma cantidad de procesos, PXp, el grado de paralelismo de ligaduras sí que se ha visto influenciado: en tanto que en la semántica mínima solamente evolucionaba una ligadura de manera local, la de la variable *main*, en la máxima también se ha dado cancha a la evolución local de los canales, teniéndose que al final PX ha sido igual a 3.

El que PMp sea mayor en la semántica máxima viene producido por el hecho de que en ambas reducciones el segundo proceso se crea en el primer paso; sin embargo, al ser mayor R en la máxima (3) que el de la mínima (2), estos dos procesos viven más, incrementándose así el grado de paralelismo medio.

Por el mero hecho de durar más la ejecución en el caso de la semántica máxima, T será mayor pero además, esta medida se ve incrementada al ser incluidas en el conjunto de ligaduras desarrollables todas las activas, los canales en nuestro caso.

Finalmente, se observa que la velocidad es inferior a la unidad, de lo que se deduce que el tiempo mayor de la semántica máxima no ha redundado en beneficios de cara a la obtención del valor principal. Puede decirse que las ligaduras que se han evaluado en la máxima eran especulativas. Y si se observan las ligaduras de ambos canales del último sistema, el hecho de que en la semántica mínima sean activas indica que el proceso creado era también especulativo.

□

Si la velocidad hubiera sido mayor que 1 tendríamos que se han evaluado ligaduras que en principio pudieron parecer especulativas, pero que a lo largo del cómputo se han revelado como necesarias para el cálculo del valor de la variable principal. Un ejemplo de este hecho se expone a continuación.

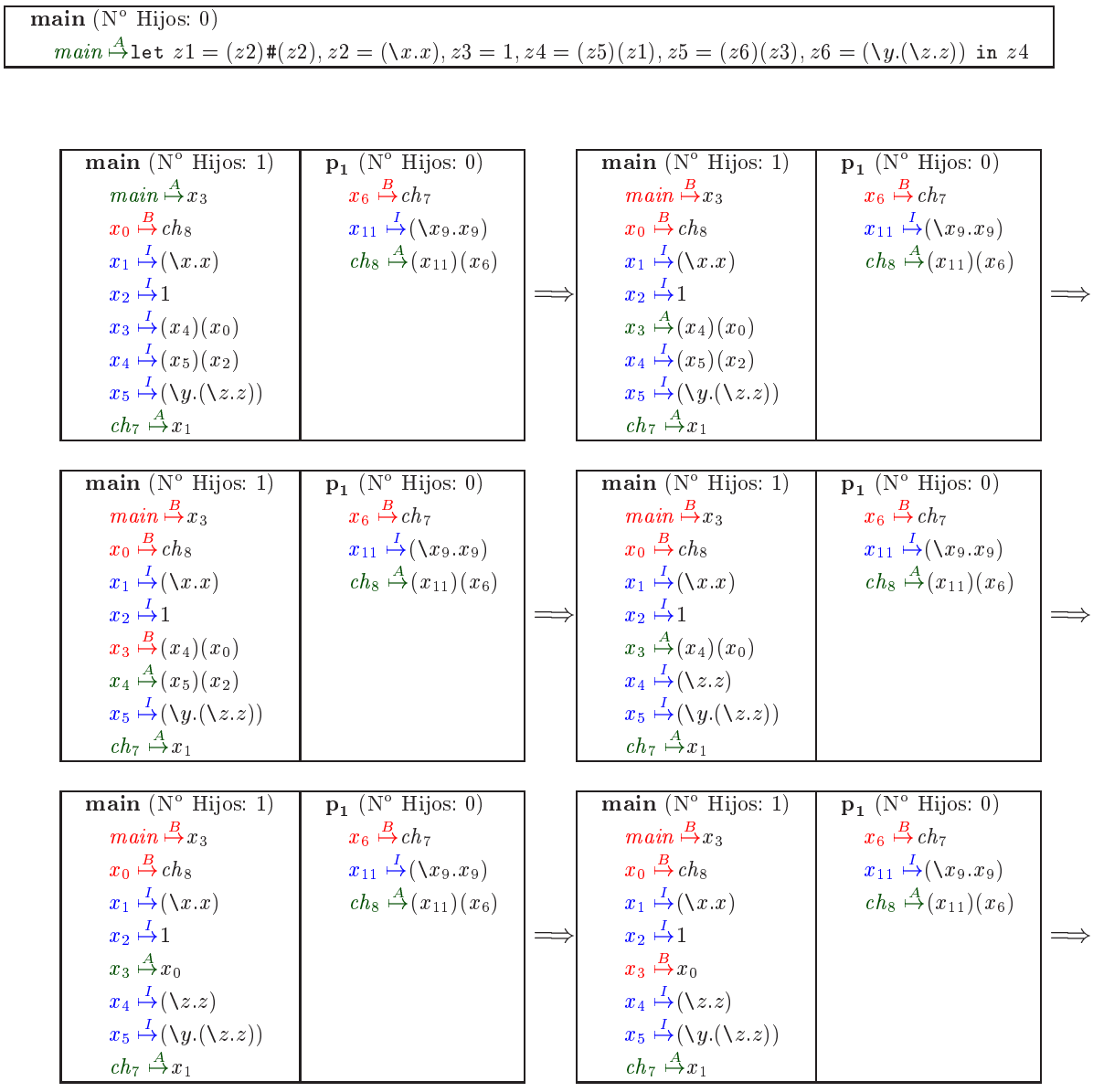
Ejemplo 6.36 *Paralelismo especulativo útil.*

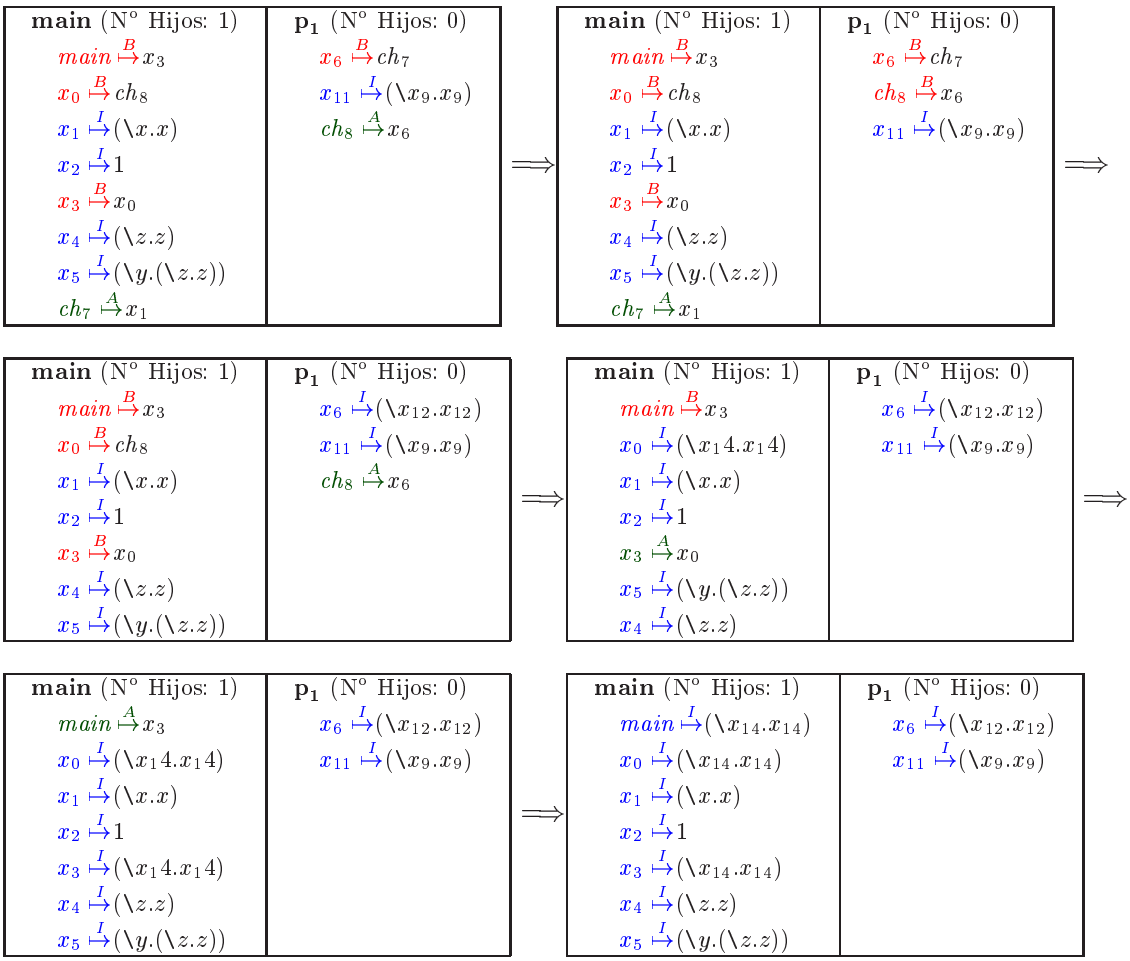
Analicemos la evaluación de la siguiente expresión:

$$\texttt{let } z_1 = (z_2)\texttt{\#}(z_2), z_2 = (\backslash x.x), z_3 = 1, z_4 = (z_5)(z_1), z_5 = (z_6)(z_3), z_6 = (\backslash y.(\backslash z.z)) \texttt{ in } z_4$$

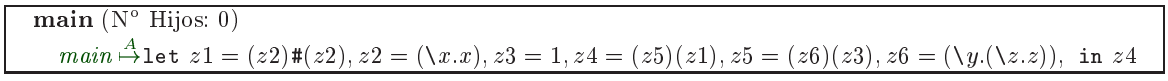
Para evaluar el cuerpo hay que llevar a cabo una doble aplicación, la segunda de las cuales va a necesitar el valor devuelto por el proceso que en principio era especulativo.

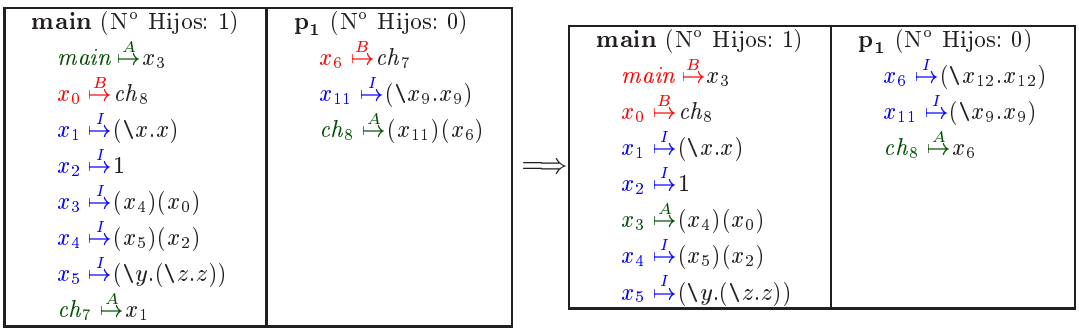
En la semántica mínima se obtiene la siguiente secuencia de reducción:

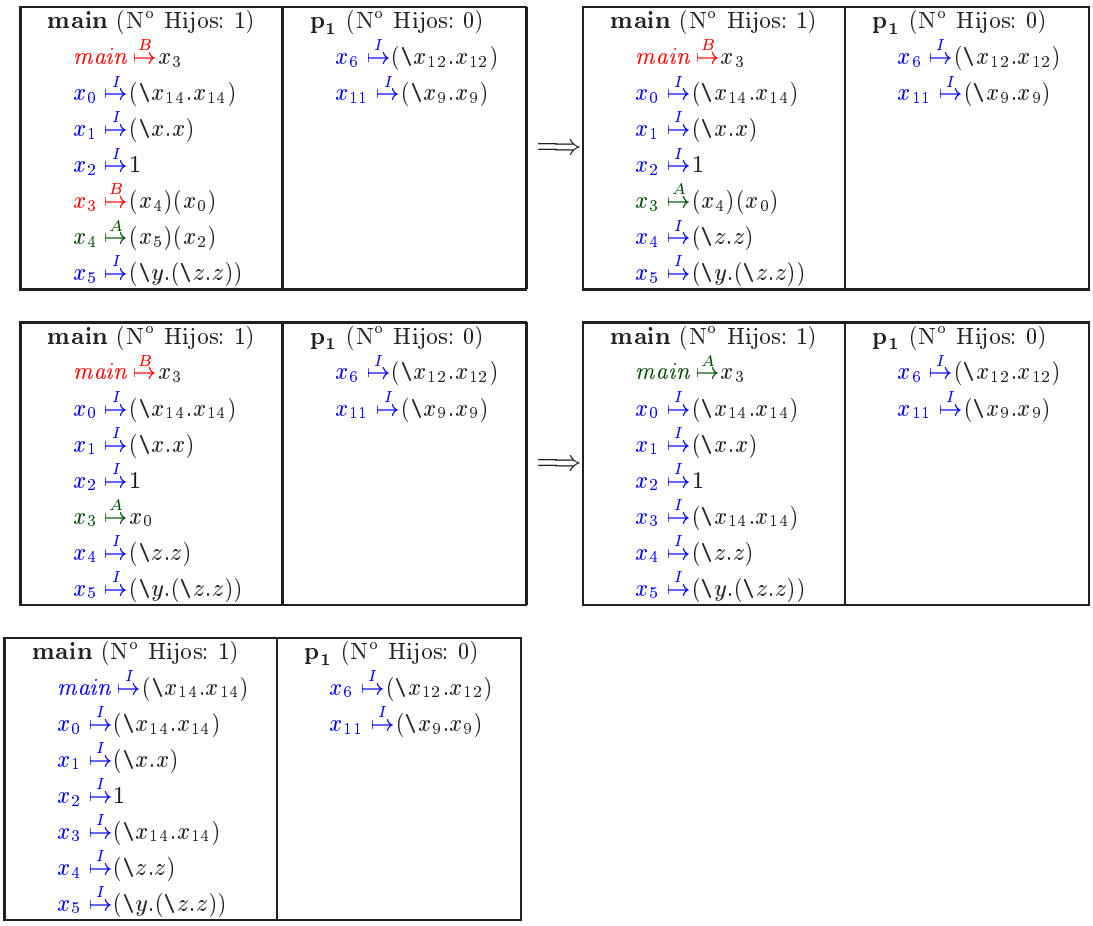




Para la semántica máxima la secuencia de reducción es la siguiente:







Una vez detalladas ambas secuencias de reducción, calculemos los valores de las medidas definidas:

	Mínima	Máxima
R	12	7
T	13	9
PM	$\frac{13}{12}$ $(1,08\hat{3})$	$\frac{9}{7}$ $(1,28\hat{5}714)$
PX	1	3
PMp	$\frac{25}{13}$ $(1,92\hat{3}076)$	$\frac{15}{8}$ $(1,875)$
PXp	2	2
V	$\frac{12}{7}$ $(1,71\hat{4}285)$	

Comparando los resultados obtenidos para ambas semánticas llegamos a las siguientes conclusiones:

1. El número de pasos de reducción en el caso máximo ha sido menor, de lo que podemos deducir que alguna de las ejecuciones en paralelo realizadas no ha sido especulativa.

2. En cuanto al número de ligaduras activas que han evolucionado, en la semántica máxima es menor. En el presente caso el proceso de demanda de una comunicación para la semántica mínima es el siguiente: una variable, x, necesita que se realice una comunicación, por lo que su ligadura se bloquea; en ese momento la ligadura del correspondiente canal se evalúa; cuando se ha obtenido el valor se comunica, se desbloquea la ligadura bloqueada de x y se continua su evaluación. Durante todo el proceso, esta ligadura ha estado dos veces activa y ha evolucionado. En la semántica máxima, cuando la ligadura de x necesita el valor que tenía que comunicarse, éste ya está disponible, por lo que directamente se toma el mismo desactivándose la ligadura en un único paso, en lugar de en dos. Este es el motivo por el cual hay menos ligaduras activas que hayan evolucionado en la semántica máxima.

3. El paralelismo máximo de ligaduras en un paso es evidentemente mayor en la semántica máxima, que ha permitido evolucionar a las ligaduras de los canales que no forman parte de la precedencia del *main*.

4. Al ser en la semántica máxima el número de pasos de reducción menor, y el momento de creación del segundo proceso el mismo en ambas, el paralelismo de procesos medio tiene que ser mayor para la máxima. En cuanto al paralelismo máximo, al no haber más procesos, es el mismo en ambos casos.

Finalmente, la velocidad es superior a 1, lo que indica que la evaluación del paralelismo especulativo compensa.

□

6.7. Semántica operacional sin evaluar copia

Hasta ahora, tanto para crear un nuevo proceso como para comunicar un valor, hemos exigido que no existieran dependencias libres sin resolver. Otra posibilidad, que va a quedar recogida en esta sección, es no evaluar las ligaduras a copiar, copiándolas en el *heap* receptor en estado inactivo. Para la presentación de esta versión alternativa de la semántica, iremos introduciendo progresivamente los cambios, siguiendo la secuencia de incorporación de estructuras sintácticas que hemos seguido en la definición anterior.

6.7.1. λ-cálculo y creación de procesos

Las transiciones locales de la Figura 6.2 siguen siendo válidas en esta versión, pues las modificaciones derivadas de la copia sin evaluar tendrán relación con la evaluación a forma normal que antes se tenía, gracias a la resolución de las dependencias libres.

Las reglas de creación de procesos (Sección 6.2.2.1) y de comunicación (Sección 6.2.2.2) varían en relación a las condiciones que deben verificarse antes de proceder:

Creación: desaparece la condición "$\mathsf{nf}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}) = \emptyset$", pues no es necesaria la resolución de dependencias de x.

Comunicación: no se exige la condición "$\mathsf{nf}(W, H_p) = \emptyset$", pues las variables libres de W no tienen necesitan estar evaluadas.

Como consecuencia, las reglas BLOCKING PROCESS CREATION, PROCESS CREATION DEMAND y VALUE COMMUNICATION DEMAND son innecesarias, pues toda creación de proceso en el nivel superior es desarrollada sin condiciones, y las demandas de dependencias libres no tienen lugar cuando las ligaduras se van a copiar, sin importar el estado de la expresión asociada.

La evaluación en paralelo no se ve afectada por el nuevo planteamiento, pues no se produce ninguna copia de ligaduras de un *heap* a otro en este ámbito local. Lo que sí es necesario cambiar es la función pre, que ahora no necesita ni el caso en el que una variable o un canal están bloqueados en una creación de proceso ni el caso en el que ch se encuentra inactivo en un valor con dependencias sin resolver.

6.7.2. Comunicación vía *streams*

Las reglas locales correspondientes a los *streams* siguen siendo válidas en el nuevo modelo de evaluación.

Sin embargo, es necesario redefinir la función nf, pues ahora solamente se van a evaluar las variables libres de las listas que haya que evaluar a forma normal, pero no las variables libres de las abstracciones.

Con esta consigna, se modifica la definición de la función nf (ver la Figura 6.20), que ya no ha de recoger tantas dependencias como en el caso precedente.

$\mathsf{nf}(E, H) \models$ (i1),(i2)-(v) $\wedge\, \forall A.(A \subseteq H \wedge A \models$ (i)-(vi) $\Rightarrow \mathsf{nf}(E, H) \subseteq A)$

(i1) $\quad x \overset{\alpha}{\mapsto} W \in H \wedge W \notin \mathit{List} \Rightarrow \emptyset \subseteq \mathsf{nf}(x, H)$

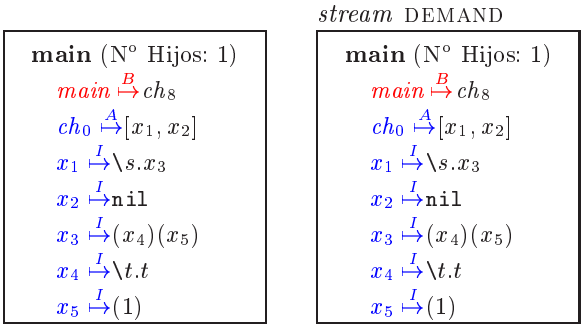
(i2) $\quad x \overset{\alpha}{\mapsto} W \in H \wedge W \in \mathit{List} \Rightarrow \mathsf{nf}(W, H) \subseteq \mathsf{nf}(x, H)$

Figura 6.20: Sin evaluar copia: detección de dependencias

En la forma descrita, las variables libres de las abstracciones no son consideradas a la hora de localizar dependencias libres sin resolver.

Ejemplo 6.37 *Demanda de la evaluación de un stream.*

El siguiente *heap* contiene una ligadura, en concreto la de ch_0, que necesita la evaluación de x_1, su cabeza. Ésta se encuentra evaluada hasta *whnf*, y sus variables libres no necesitan ser demandadas, por lo que la regla STREAM DEMAND no tiene ningún efecto:



□

6.7.2.1. Comunicación

Para la nueva versión de evaluación hay que definir unas nuevas reglas globales de comunicación, que quedan recogidas en la Figura 6.21.

(value communication)

$$\text{si } (W \in \mathit{List} \Rightarrow W \equiv \texttt{nil})$$
$$\mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{vCom}{\longrightarrow}$$
$$\overset{vCom}{\longrightarrow} (S, \langle p, H_p\rangle, \langle c, H_c + \eta(\mathsf{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} \eta(W)\}\rangle)$$

(head-stream communication)

$$H'_p = H_p + \{ch \overset{\alpha}{\mapsto} [x_1 : x_2], x_1 \overset{I}{\mapsto} W\}$$
$$\text{si}(\mathsf{nf}(W, H'_p) = \emptyset) \wedge (\mathsf{isB}(x_1, H'_p))$$
$$\mathit{fresh}(y_1, y_2), \mathit{freshrenaming}(\eta)$$
$$(S, \langle p, H_p + \{ch \overset{\alpha}{\mapsto} [x_1 : x_2], x_1 \overset{I}{\mapsto} W\}\rangle, \langle c, H_c + \{\theta \overset{B}{\mapsto} ch\}\rangle) \overset{sCom}{\longrightarrow}$$
$$\overset{sCom}{\longrightarrow} (S, \langle p, H_p + \{ch \overset{A}{\mapsto} x_2, x_1 \overset{I}{\mapsto} W\}\rangle, \langle c, H_c + \eta(\mathsf{nh}(W, H_p)) + \{\theta \overset{A}{\mapsto} [y_1 : y_2], y_1 \overset{A}{\mapsto} \eta(W), y_2 \overset{B}{\mapsto} ch\}\rangle)$$

Figura 6.21: Sin evaluar copia: reglas globales de comunicación

La regla de comunicación VALUE COMMUNICATION varía con respecto a la de la Figura 6.9, en tanto los valores que se comunican ya no necesitan estar en forma normal, por lo que desaparece la condición sobre las dependencias libres.

En el caso de comunicación de la cabeza de un *stream*—llevada a cabo por la regla HEAD-STREAM DEMAND— dicha cabeza tiene que ser un valor en forma normal, hecho que obliga a detectar todas las dependencias libres de la cabeza y a resolverlas antes de proceder con la comunicación. Bajo estas condiciones, la comunicación procede igual que en la regla correspondiente al caso de copia evaluada, presentada en la Figura 6.9.

Ilustremos el funcionamiento de ambos tipos de comunicación mediante un par ejemplos.

Ejemplo 6.38 *Comunicación de un valor simple.*

En el *heap* inferior ch_0 está ligada a un valor *whnf* que se procede a comunicar. Observamos que este valor contiene una variable libre ligada a una expresión que no es un valor; pero esto no supone un impedimento a la comunicación, que procede copiando esta ligadura tal y como se encuentra, exceptuando su renombramiento.

main (N° Hijos: 1)	**main.1** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{B}{\mapsto} ch_0$
$x_2 \overset{I}{\mapsto} (x_3)(x_4)$	$ch_8 \overset{B}{\mapsto} x_7$
$x_3 \overset{I}{\mapsto} \backslash x_5.x_5$	
$x_4 \overset{I}{\mapsto} 3$	
$x_6 \overset{B}{\mapsto} ch_8$	
$ch_0 \overset{I}{\mapsto} \backslash x_1.x_2$	

VALUE COMMUNICATION

main (N° Hijos: 1)	**main.1** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_7 \overset{A}{\mapsto} \backslash x_9.x_{10}$
$x_2 \overset{I}{\mapsto} (x_3)(x_4)$	$x_{10} \overset{I}{\mapsto} (x_{11})(x_{12})$
$x_3 \overset{I}{\mapsto} \backslash x_5.x_5$	$x_{11} \overset{I}{\mapsto} \backslash x_{13}.x_{13}$
$x_4 \overset{I}{\mapsto} 3$	$x_{12} \overset{I}{\mapsto} 3$
$x_6 \overset{B}{\mapsto} ch_8$	$ch_8 \overset{B}{\mapsto} x_7$

Como el cometido de ch_0 ha sido cumplido, éste desaparece del sistema.

□

Ejemplo 6.39 *Comunicación de la cabeza de un stream.*

En el *heap* inferior ch_0 está ligada a un *stream* cuya cabeza es un valor que no es una lista y que puede ser comunicado.

main (N° Hijos: 1)	**main.1** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_8 \overset{B}{\mapsto} ch_0$
$x_1 \overset{I}{\mapsto} (\backslash x.x_3)$	$ch_9 \overset{B}{\mapsto} x_7$
$x_2 \overset{I}{\mapsto} \texttt{nil}$	
$x_3 \overset{I}{\mapsto} [x_4 : x_2]$	
$x_4 \overset{I}{\mapsto} (x_5)(x_6)$	
$x_5 \overset{I}{\mapsto} \backslash y.y$	
$x_6 \overset{I}{\mapsto} 5$	
$x_7 \overset{B}{\mapsto} ch_8$	
$ch_0 \overset{I}{\mapsto} [x_1 : x_2]$	

Se procede a la comunicación, con la correspondiente copia de dependencias libres del valor comunicado.

HEAD-STREAM COMMUNICATION

main (N° Hijos: 1)	**main.1** (N° Hijos: 0)
$main \overset{B}{\mapsto} x_6$	$x_8 \overset{A}{\mapsto} [x_{10} : x_{11}]$
$x_1 \overset{I}{\mapsto} (\backslash x.x_3)$	$x_{10} \overset{A}{\mapsto} (\backslash x.x_{12})$
$x_2 \overset{I}{\mapsto} \texttt{nil}$	$x_{11} \overset{B}{\mapsto} ch_0$
$x_3 \overset{I}{\mapsto} [x_4 : x_2]$	$x_{12} \overset{I}{\mapsto} [x_{13} : x_{14}]$
$x_4 \overset{I}{\mapsto} (x_5)(x_6)$	$x_{13} \overset{I}{\mapsto} (x_{15})(x_{16})$
$x_5 \overset{I}{\mapsto} \backslash y.y$	$x_{14} \overset{I}{\mapsto} \texttt{nil}$
$x_6 \overset{I}{\mapsto} 5$	$x_{15} \overset{I}{\mapsto} \backslash z.z$
$x_7 \overset{B}{\mapsto} ch_8$	$x_{16} \overset{I}{\mapsto} 5$
$ch_0 \overset{A}{\mapsto} x_2$	$ch_9 \overset{B}{\mapsto} x_7$

□

La comunicación de un valor, ya sea un valor único o la cabeza de un *stream*, puede dar lugar a nuevas comunicaciones. Por ello, las dos reglas de la Figura 6.21 tienen que ser aplicadas intercalada y repetidamente hasta que no sea posible ninguna comunicación más. De esta forma se define la regla de múltiple paso para comunicaciones:

$$\overset{Com}{\Longrightarrow} = (\overset{sCom}{\Longrightarrow} \circ \overset{vCom}{\Longrightarrow})^*.$$

6.7.2.2. Planificación

Recordemos que el cambio de etiquetas supone realizar:

1. El desbloqueo de las ligaduras que dependían de una variable ya asociada a valor *whnf*.
2. La desactivación de las ligaduras que ya han alcanzado una *whnf*.
3. La demanda de la evaluación de las ligaduras necesarias para llevar a cabo comunicaciones pendientes.

Estas funciones son llevadas a cabo por algunas de las reglas presentadas Figura 6.6, a las que añadimos la nueva regla global de la Figura 6.22.

El desbloqueo y la desactivación de ligaduras se realiza en la forma que se especificó en la Figura 6.6.

En cambio, la demanda para comunicaciones varía, pues hay que distinguir entre la comunicación de un valor simple y la comunicación de la cabeza de un *stream*. En el primer caso hemos visto que en cuanto el canal está ligado a un *whnf*, se está en disposición de comunicar dicho valor, por lo que no hace falta una regla de demanda. En el segundo caso, la cabeza del *stream* está casi lista para ser comunicada, aunque falta

(head-stream communication demand)

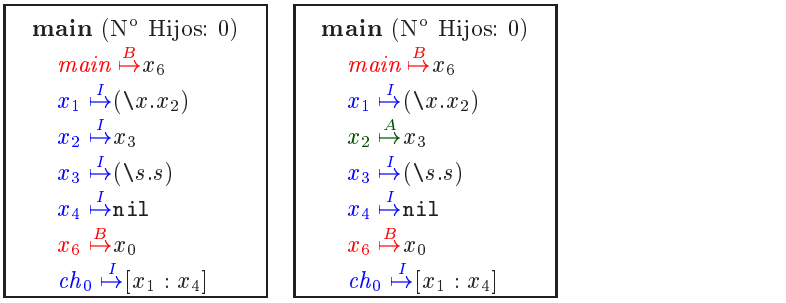
$$H' = H + \{x_1 \overset{I}{\mapsto} W,\ ch \overset{I}{\mapsto} [x_1 : x_2]\}$$
$$\text{si } x \overset{I}{\mapsto} E \in \mathsf{nf}(W, H)$$
$$(S, \langle p, H + \{x_1 \overset{I}{\mapsto} W,\ ch \overset{I}{\mapsto} [x_1 : x_2]\}\rangle) \overset{hsComd}{\longrightarrow} (S, \langle p, H + \{x_1 \overset{I}{\mapsto} W,\ ch \overset{I}{\mapsto} [x_1 : x_2], x \overset{A}{\mapsto} E\}\rangle)$$

Figura 6.22: Sin evaluar copia: planificación

resolver algunas de sus dependencias para que la cabeza comunicada se encuentre en forma normal; la regla HEAD-STREAM COMMUNICATION DEMAND se encarga de demandar estas dependencias.

Ejemplo 6.40 *Demanda para comunicar la cabeza de un stream.*

En el siguiente *heap* x_1 está ligada a un valor en forma normal, que se podría comunicar. Sin embargo, depende de la variable libre x_2, aún sin evaluar. Se procede entonces a la activación de la ligadura de x_2:



□

De esta forma, las reglas que regulan el proceso de desbloqueo, desactivación y demandas globales del subconjunto Jauja básico con *streams* son: WHNF UNBLOCKING, WHNF DEACTIVATION, y HEAD-STREAM COMMUNICATION DEMAND, que se combinan en la transición de paso múltiple:

$$\overset{Unbl}{\Longrightarrow} = \overset{hsComd}{\Longrightarrow} \circ \overset{wDeact}{\Longrightarrow} \circ \overset{wUnbl}{\Longrightarrow}.$$

El resto de extensiones —no determinismo y canales dinámicos— no ven afectado su funcionamiento porque no se fuerce la evaluación de las ligaduras que han de ser copiadas.

PARTE III

SEMÁNTICA DENOTACIONAL

CAPÍTULO 7

Continuaciones

> Cuando una idea es inventada varias veces...
> ...es porque ha llegado su tiempo.
>
> Roger Hindley, Samson Abramsky

Según el diccionario de María Moliner [Mol96], *continuación* tiene dos acepciones:

1. Acción de continuar: "Se ha tratado de la continuación de las obras del ferrocarril".
2. Cosa o parte con que continúa algo: "La continuación de la novela".

Ambos significados toman relevancia dentro del ámbito de las Ciencias de la Computación: el término *continuación* define una función que desempeña su misión tras la evaluación de una expresión o de una instrucción, llevando a cabo el resto —la continuación— del cómputo que se tenía que realizar cuando dicha expresión o instrucción hubiera concluido sus tareas.

Vamos a introducir por etapas esta noción de continuación, viendo primero cómo se generó.

7.1. Los descubrimientos de las continuaciones

La definición del concepto de continuación fue como el descubrimiento de América, que fue encontrado en diversas ocasiones, en parte porque no hubo comunicación entre

los descubridores previos.[1] Aunque, según Reynolds [Rey93], la causa de estos múltiples descubrimientos se debe menos a la pobre comunicación entre científicos en aquellos tiempos y más a la gran variedad de situaciones en las cuales las continuaciones resultaron ser útiles. El factor común de todos los elementos precursores de las continuaciones, así como de sus descubrimientos, es la necesidad de llevar cuenta de las tareas que hay que llevar a cabo tras la ejecución o desarrollo de una parte de código, de una expresión, etc. Procedamos a recorrer el camino que se describe en [Rey93].

Durante los años sesenta, la implementación y la definición formal de lenguajes de programación fueron precursores del descubrimiento de las continuaciones. *Naur* [Nau63], en su intención de especificar el cambio en el control de los programas — como consecuencia de saltos, llamadas a procedimientos, etc.— necesitaba suministrar una descripción estática del destino del control, así como una descripción dinámica del entorno y la referencia a la pila, elementos básicos de la máquina que ejecutaba los cómputos. Ambas descripciones conformaban un "punto de programa", que puede considerarse una representación de una continuación. *Dijkstra* [Dij60] se preocupó por cómo debía seguir la evaluación de un programa tras la ejecución de una subrutina. Los datos necesarios para esta tarea los agrupó en un *enlace*, que puede considerarse también como una continuación. Finalmente, *Landin* [Lan64] definió la máquina SECD (Stack, Environment, Control list, Dump) para interpretar lenguajes de expresiones aplicativas, con orden de evaluación *call-by-value* (definida en el glosario del Apéndice B); este intérprete disponía de un estado formado por cuatro componentes:

(pila, entorno, lista de control, depósito).

El depósito codificaba el resto del cómputo que tenía que realizarse cuando el control estaba agotado; en definitiva, era otra representación de una continuación. Por otra parte, para traducir los saltos incondicionales (`goto`), Landin definió el operador J [Lan65a, Lan65b], que recogía en una clausura de programa el depósito que la máquina tenía en ese momento; dicha clausura podría ser invocada más tarde para restaurar el depósito capturado. Estos son los primeros pasos impulsores de las continuaciones, pero su reiterado descubrimiento explícito se produjo un poco más tarde.

Van Wijngaarden, en 1964, dio la primera descripción del uso de continuaciones en una conferencia que, posteriormente, fue recogida en [Wij66]. Su trabajo consistió en formular un preprocesador que traduciría Algol 60 a un sublenguaje más restringido, sin saltos incondicionales ni etiquetas, siendo el final del preproceso una transformación de procedimientos en un *continuation passing style (CPS)* (ver Apéndice B).

En diciembre de 1969, *Mazurkiewicz* hizo gernimar la idea de las *funciones de cola* [Maz71], un concepto de algoritmo similar al de los autómatas y compuesto por un

[1] Puede considerarse que América fue descubierta por primera vez cuando sus primeros pobladores llegaron allí, 30.000-20.000 a.C. El segundo descubrimiento, a finales del siglo X d.C., fue obra del pueblo normando (vikingo). El tercero estuvo en manos del Imperio Chino, con anterioridad al siglo XV d.C. en sus exploraciones marítimas por Sudamérica y Australia y muy probablemente también alcanzó el continente americano el explorador Zheng He en el siglo XV d.C. [Hsu04]. Finalmente, el cuarto y más conocido es el de Cristóbal Colón, en 1492 d.C.

conjunto de etiquetas —un subconjunto de las mismas que conformaban las terminales— un conjunto de estados y una función parcial de transición que desde un estado, y con una etiqueta no terminal, conducía a un estado y a una etiqueta posiblemente terminal. La semántica de una de estas funciones era un entorno que asociaba etiquetas con continuaciones de instrucción, en el estilo de las etiquetas y saltos incondicionales de los lenguajes imperativos. Estas ideas inspiraron posteriormente a Wadsworth, como veremos más adelante.

El siguiente descubrimiento lo realizó *F. L. Morris* en las postrimerías del año 1970. Su trabajo se centró en el diseño de intérpretes definicionales para un lenguaje funcional con *call-by-value* (definida en el glosario del Apéndice B) y extensiones que incluían asignaciones y etiquetas [Mor93]. En el intérprete final las continuaciones se usaban para el tratamiento de etiquetas y saltos: específicamente, los depósitos de Morris eran continuaciones de expresión y los valores de etiqueta eran continuaciones de instrucción. Las ideas de Morris fueron el fundamento para los trabajos de *Reynolds* [Rey72]: éste empleó en la construcción de interpretes definicionales una representación explícita de las continuaciones para eliminar las dependencias del lenguaje definido, con respecto a la técnica de paso de parámetros del lenguaje usado para definirlo.

Y llegamos al descubrimiento de las continuaciones en el marco de la semántica denotacional. Los prolíficos —en el tema de continuaciones— finales del año 1970 acogieron el descubrimiento de *Wadsworth* del uso de las continuaciones para dotar de semántica a etiquetas y saltos incondicionales, observando que este mecanismo también era válido para dar significado al *call-by-value* y a otras construcciones que restringían el orden de evaluación. Fue precisamente este autor el que acuñó el nombre de continuaciones. Una versión amplia de sus ideas aparece en [SW00].

Ya en 1971, *J. H. Morris* describe una transformación CPS para programas escritos en un lenguaje del tipo de Algol. Una transformación parecida a la de Van Wijngarden con un correcto tratamiento de los bloques anidados, y en la que funciones y procedimientos son tratados como tales, en lugar de transformar, en un paso previo, las primeras en un caso particular de los segundos.

En los albores del siguiente año, *Fischer* extiende el λ-cálculo *call-by-value* con expresiones condicionales y constantes y funciones primitivas sin interpretar, y describe una transformación de este lenguaje funcional en un CPS (versión más completa de sus trabajos en [Fis93]). Asimismo, afirma que, usando CPS, cualquier programa puede ser transformado en una forma que puede ser implementada por una pila. Fisher es el primero que da una demostración sobre la semántica de continuaciones, asegurando que la transformación CPS preserva el significado en un sentido apropiado.

Nuestro recorrido por los descubrimientos de las continuaciones tiene su última estación en los trabajos de *Abdali* [Abd73]. Este autor describió una nueva forma de definición de lenguaje, en la que los programas escritos en Algol 60 eran traducidos al λ-cálculo sin tipos. Para tratar los aspectos imperativos del lenguaje, ideó una traducción muy parecida a la transformación CPS.

El paseo, plagado de continuaciones, nos ha llevado por tres caminos:

- Un método de transformación en CPS.
- Un estilo de interpretes definicionales, que define un lenguaje por medio de un intérprete escrito en otro lenguaje.
- Un estilo de semántica denotacional, en el sentido de Scott y Strachey [Sto77].

Es este tercer camino el que centra nuestro interés dentro del presente trabajo. El segundo pretendemos analizarlo implementando la semántica denotacional de continuaciones que definimos para Jauja.

7.2. Continuaciones y semántica denotacional

¿Cuál puede ser el interés de no definir una semántica denotacional directa, en la que a cada instrucción o expresión se le asocia directamente su valor denotacional, sin considerar el posible contexto en el que se encuentra, y componer en su lugar una semántica denotacional basada en continuaciones? Diversas respuestas se han dado a lo largo de la breve, pero productiva, Historia de las Ciencias de la Computación. Ya se han esbozado en la Sección 7.1 los diferentes campos en los que surgieron las continuaciones, y se ha mencionado que en el de la semántica denotacional habían sido empleadas para dar significado a algunas construcciones sintácticas; pero veamos ahora algunos argumentos más contundentes que justifican su uso.

Strachey y Wadsworth describen en [SW00] (y Stoy recoge en [Sto77]) la incapacidad de una semántica denotacional directa para poder expresar la ruptura en la continuidad de la evaluación de una composición secuencial de instrucciones. La forma habitual de definir el significado de una instrucción, Γ, es:

$$\mathcal{C}\,[\![\Gamma]\!]\,\rho = \theta$$

donde ρ es el entorno en el que se definen las variables libres de la instrucción y θ es una función que dado un estado devuelve otro. Entonces la semántica de una composición secuencial de instrucciones se expresa por medio de:

$$\mathcal{C}\,[\![\Gamma_1; \Gamma_2]\!]\,\rho = (\mathcal{C}\,[\![\Gamma_2]\!]\,\rho) \circ (\mathcal{C}\,[\![\Gamma_1]\!]\,\rho).$$

Pero en esta definición no se da posibilidad a Γ_1 de decidir si Γ_2 debe ser obedecido o no, salvo en los casos de no-terminación o parada por error, en los que la estrictez del operador de composición funcional lleva a devolver el valor indefinido. Según Strachey y Wadsworth, para poder utilizar la semántica directa, la solución pasaría por una transformación sintáctica adecuada que eliminara convenientemente los saltos y convirtiera el programa en un elemento secuencial. Sin embargo, esta solución no es fácil cuando el salto no es sintácticamente evidente, por ejemplo en una llamada del tipo `call` I.

Transformaciones como las realizadas por Van Wijngaarden en [Wij66] podrían solucionar el problema, pero el enfoque de Scott y Strachey no es partidario de esta solución, pues ellos tratan de expresar la semántica de un programa en base a la semántica de sus componentes según vienen inicialmente escritas. Para ello deben realizar, por medio de la semántica las transformaciones, que Van Wijngaarden realiza en el nivel sintáctico.

La solución es abandonar la idea de devolver solamente una función transformadora de estados, y admitir que el contexto es importante para evaluar tanto una instrucción como una expresión. El comportamiento del contexto tendrá que ser pasado como argumento a $\mathcal{C}\,[\![\Gamma]\!]\,\rho$. Este es el punto donde aparecen las continuaciones: el valor denotacional de una instrucción será ahora una función que tendrá un parámetro más, la continuación θ, quedando la llamada $\mathcal{C}\,[\![\Gamma]\!]\,\rho\,\theta$. Esta función θ será una transformadora de estados y recogerá el efecto del resto del programa, esto es del contexto en el que se encuentra. Con esta idea, la semántica de la composición secuencial de instrucciones se expresaría:

$$\mathcal{C}\,[\![\Gamma_1;\Gamma_2]\!]\,\rho\,\theta = \mathcal{C}\,[\![\Gamma_1]\!]\,\rho(\mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$$

de modo que la ejecución de $\Gamma_1;\Gamma_2$ con la continuación (de instrucción) θ conlleva ejecutar Γ_1, siendo su continuación la ejecución de Γ_2 con la continuación inicial θ. De este modo, el valor de $\mathcal{C}\,[\![\texttt{goto}\ E]\!]\,\rho\,\theta$ se puede definir sin que dependa de la continuación θ. Idéntico argumento es esgrimido por Reynolds en [Rey98], y por Morris en [Mor93]:

> The problem with labels is that the evaluation of any subexpression may result in going to one, in which case the computation which might have been planed on to complete the evaluation of the main expression will have to be forgotten about. This means that the function compiled for the subexpression should be passed as an argument something which says what more is waiting to be done, so that the sub-expression can decide whether to do it or not.[2]

Por otro lado, en [Jos89] se argumenta que si bien la semántica presentada en [Sto77] distingue *call-by-value* y *call-by-name*, no lo hace con *call-by-name* y *call-by-need*. Gracias a las continuaciones de expresión empleadas por Josephs se consigue un modelo denotacional para lenguajes funcionales perezosos. Pero, ¿cuál es la similitud entre una composición secuencial de instrucciones y la reducción *call-by-need*? Hemos visto la afición de Γ_1 por romper la secuencia y no obedecer a Γ_2. Siguiendo *call-by-need*, el argumento, E_2, de una aplicación funcional, E_1E_2, solamente se evalúa si es estrictamente necesario, necesidad que se decide evaluando el cuerpo de la abstracción resultante de E_1, tras una adecuada sustitución del parámetro formal por el argumento. En definitiva, la primera parte de la aplicación puede decidir si obedecer y evaluar E_2 o no. Por otra parte, el uso del resultado de evaluar E_1 viene determinado por el contexto en el que

[2] El problema con las etiquetas es que la evaluación de cualquier subexpresión puede que lleve a una de ellas (etiquetas), en cuyo caso el cómputo que podría haber sido planificado para la completa evaluación de la expresión principal tendrá que ser olvidado. Es decir, la función compilada para la subexpresión debería ser pasada como un argumento que dijera qué más está esperando a ser realizado, de manera que la subexpresión pueda decidir si hacerlo o no.

se enmarca: en el caso de encontrarse inmersa en una aplicación funcional ese resultado tiene que ser aplicado al argumento, pero si E_1 conformara un programa completo, la acción que reste será independiente del contexto (vacío) pudiéndose limitar, por ejemplo, a imprimir su resultado o, simplemente, no realizar acción alguna.

Ya hemos mencionado los dos tipos de continuación que se usan en el ámbito de este tipo de semántica denotacional: de instrucción y de expresión. La coexistencia de ambos tipos tiene su razón de ser en la distinta naturaleza que tienen instrucciones y expresiones. La evaluación de las primeras solamente produce cambios en el estado de la máquina; la evaluación de una expresión, además de una modificación del mismo tipo, devuelve un resultado, y lo que va a hacerse con este resultado no viene determinado por la expresión evaluada, sino que es el contexto —como hemos visto en el caso de la aplicación funcional *call-by-need*— quien precisa el futuro uso del resultado. Tennent [Ten76] describe el tipo más general que puede tener cada una de ellas:

$$\mathbf{Cont} = \mathbf{Estado} \to \mathbf{Respuesta}$$
$$\mathbf{ECont} = \mathbf{EVal} \to \mathbf{Estado} \to \mathbf{Respuesta}.$$

Observamos que la diferencia estriba en el parámetro adicional que tienen las continuaciones de expresión; en ellas una vez que una continuación de expresión toma el valor resultado de evaluar la expresión, el comportamiento es idéntico al de una continuación de instrucción.

$$\mathbf{Cont} = \mathbf{Estado} \to \mathbf{Respuesta}$$
$$\mathbf{ECont} = \mathbf{EVal} \to \mathbf{Cont}.$$

En el caso general, una continuación de instrucción transforma un estado en una respuesta que puede ser de cualquier tipo. Lo más habitual es que la **Respuesta** sea un **Estado**, siendo entonces una continuación la función transformadora de estados:

$$\mathbf{Cont} = \mathbf{Estado} \to \mathbf{Estado}$$
$$\mathbf{ECont} = \mathbf{EVal} \to \mathbf{Cont}.$$

Y dependiendo de la definición del estado se dotará de significado a distintas características de los lenguajes de programación. Por ejemplo, la memoria de datos de los lenguajes imperativos (esquema en Figura 7.1) se divide en dos partes: el entorno que liga cada variable de programa a una dirección de memoria, y el almacén donde cada dirección de memoria contiene un valor. El entorno varía dependiendo del ámbito de las variables; en el caso de un procedimiento, cuando éste es llamado, el entorno liga los nombres de variable a las direcciones correspondientes en ese entorno; esta asociación se desliga cuando la ejecución de la llamada termina, pudiéndose, de esta manera, programar con la nada recomendable práctica de usar los mismos nombres de variable en distintas ocasiones.

Pero mientras que el entorno se construye de manera local para cada instrucción o expresión a evaluar, el almacén de direcciones de memoria es el mismo siempre, realizándose sobre este almacén único las actualizaciones y nuevas reservas oportunas. Por ello, en los modelos habituales de continuaciones, el entorno es un argumento que se

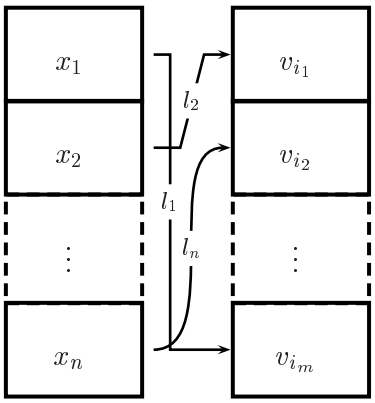


Figura 7.1: Memoria imperativa

pasa a la evaluación de la expresión fuera del estado:

$$\mathcal{C} :: \mathbf{Ins} \to \mathbf{Env} \to \mathbf{Cont} \to \mathbf{Cont}$$
$$\mathcal{E} :: \mathbf{Exp} \to \mathbf{Env} \to \mathbf{ECont} \to \mathbf{Cont}.$$

Tiene sentido dar el tipo de $\mathcal{E}$ *currificado* (ver Apéndice B), porque cada aplicación parcial tiene un significado intuitivo:

$\mathcal{C}[\![\Gamma]\!]$, $\mathcal{E}[\![E]\!]$*:* es el significado de la instrucción/expresión "en vacío";

$\mathcal{C}[\![\Gamma]\!]\,\rho$, $\mathcal{E}[\![E]\!]\,\rho$*:* ya disponen del entorno, y su significado es independiente de la continuación (de expresión) o del estado que se les pase como argumentos. Habitualmente, los valores de función se encuentran en este tipo;

$\mathcal{C}[\![\Gamma]\!]\,\rho\,\theta$, $\mathcal{E}[\![E]\!]\,\rho\,\kappa$*:* representan un cómputo en el que todo está listo, pues se tiene la información del contexto donde se encuentra la instrucción Γ/la expresión E. Los valores de las etiquetas serían de este tipo;

$\mathcal{C}[\![\Gamma]\!]\,\rho\,\theta s$, $\mathcal{E}[\![E]\!]\,\rho\,\kappa\,s$*:* corresponden a una ejecución particular, disponiendo de un estado particular.

Hemos tratado el procedimiento para evaluar con continuaciones, pero no se ha mencionado la llamada inicial con una expresión-programa inicial. Para ello se necesita un entorno inicial, ρ_0, que puede ser vacío —ningún identificador está asociado aún a ninguna dirección—, o contener las asociaciones necesarias para poder definir las funciones primitivas del lenguaje. En cuanto a la continuación inicial, existen diversas propuestas: en [Sto77] se propone la *identidad*, que en el supuesto de continuaciones de expresión ante cualquier valor dado devuelve el estado intacto; sin embargo, en [Jos89] se propone la continuación de expresiones inicial *print*, que imprime en la salida el valor argumento.

Nosotros vamos a seguir los principios de una semántica denotacional con continuaciones para definir formalmente el significado de Jauja.

CAPÍTULO 8

Semántica denotacional

> El verdadero significado de las cosas se encuentra al decir las mismas cosas con otras palabras.
>
> Charles Chaplin

La presentación de la semántica denotacional para Jauja se va a realizar en dos formas: inicialmente se definirá el significado formal del núcleo básico con no-determinismo simple; posteriormente se realizará lo mismo para este mismo subconjunto tras añadirle la comunicación vía *streams*. De esta forma podremos ir observando más nítidamente las peculiaridades de cada construcción.

El modelo denotacional elegido no es una semántica denotacional directa, sino un modelo de continuaciones. Esta decisión viene determinada por la necesidad de expresar la pereza de Jauja y los posibles *efectos laterales* producidos como resultado de la evaluación de una expresión. Por ejemplo, si con Jauja queremos evaluar la expresión $x_1 \# x_2$, el valor devuelto como resultado sería el que proviene de aplicar a x_2 el valor resultante de x_1. Sin embargo, la intención de la semántica que se presenta en este capítulo no es dar a una expresión como único valor denotacional el valor producido, sino mostrar explícitamente el paralelismo que da sentido a la existencia de Jauja. Por ello, además de obtener el valor correspondiente a $x_1 \# x_2$, se tiene que crear un proceso, considerándose esta creación y las comunicaciones subyacentes como efectos laterales de la devolución de un valor. Por otra parte, se dan casos en los que el efecto lateral depende de la pereza de una expresión; por ejemplo, dada la expresión (sin normalizar)

$(\backslash x.(1))((\backslash z.z)((\backslash y.y)\#\,4))$, la creación de proceso no debe figurar en el valor denotacional asociado a ella; una semántica directa que calculase los valores denotacionales de ambas subexpresiones, $\backslash x.(1)$ y $(\backslash y.y)\#\,4$, incluiría las denotaciones de la creación y de las comunicaciones correspondientes, sin que tenga sentido ninguna de ellas por no ser demandada la evaluación del argumento de la aplicación; la decisión de incluir o no estos efectos laterales la llevará implícita la continuación.

Antes de comenzar las correspondientes descripciones de dominios y funciones semánticas, cabe decir que, como en Jauja no se tienen instrucciones, la función de evaluación que definiremos será $\mathcal{E}$, correspondiente a la evaluación de expresiones.

8.1. Jauja básico con no-determinismo explícito simple

En esta sección definimos la semántica denotacional del subconjunto de Jauja, formado por el λ-cálculo básico, la declaración local de variables y la creación de procesos, al que se le añade no-determinismo explícito simple, que en lugar de mezclar dos listas en una sola, escoge de manera no-determinista entre dos valores de abstracción. Se muestra así cómo se estructura el modelo para tratar el no-determinismo pero sin incorporar por el momento listas y *streams*, lo que dificultaría la presentación de los conceptos más básicos.

El proceso de construcción de una semántica denotacional pasa por la definición de los dominios semánticos y de la función semántica de evaluación.

8.1.1. Dominios semánticos

En una semántica de continuaciones el significado de un programa es una transformación de estados, por lo que un dominio de continuaciones será la base de los dominios. Para poder definirlo se necesita construir primero el dominio de estados. En el caso de las continuaciones de expresión,[1] aparte de los estados, son necesarios los valores a los que dan lugar las expresiones, pues dependiendo de ellos se construirá la correspondiente continuación.

El conjunto de dominios, que pasamos a explicar seguidamente, aparece en la Figura 8.1, donde $\mathcal{P}_f(A)$ denota el conjunto de partes finitas del conjunto A.

Como en esta sección Jauja presenta no-determinismo, una *continuación*, **Cont**, no transforma un estado, **State**, en otro, sino que un estado puede transformarse en varios estados diferentes, **SState**. Como repetidamente hemos mencionado, una *continuación de expresión*, **ECont**, tomará un valor, derivado de la evaluación de una expresión, y

[1] Para simplificar la nomenclatura, denominaremos *continuación* a una función transformadora de estados y tendremos una *continuación de expresión* cuando, previamente a la transformación de estados, es necesario haber obtenido un valor.

	Cont	=	**State** $\to$ **SState**	continuaciones
$S \in$	**SState**	=	$\mathcal{P}_f$(**State**)	conjuntos de estados
$s \in$	**State**	=	**Env** $\times$ **SChan**	estados
$\kappa \in$	**ECont**	=	**EVal** $\to$ **Cont**	continuaciones de expresión
$\rho \in$	**Env**	=	**Ide** $\to$ (**Val** + {undefined})	entornos
$v \in$	**Val**	=	**EVal** + (**IdProc** $\times$ **Clo**) + {not_ready}	valores
$\varepsilon \in$	**EVal**	=	**Abs** $\times$ **Ides**	valores de expresión
$\alpha \in$	**Abs**	=	**Ide** $\to$ **Clo**	valores de abstracción
$\nu \in$	**Clo**	=	**IdProc** $\to$ **ECont** $\to$ **Cont**	clausuras
sch $\in$	**SChan**	=	$\mathcal{P}_f$(**Chan**)	conjuntos de canales
	Chan	=	**IdProc** $\times$ **CVal** $\times$ **IdProc**	canales
	CVal	=	**EVal** + {unsent}	valores comunicables
$I \in$	**Ides**	=	$\mathcal{P}_f$(**Ide**)	conjuntos de identificadores
$p, q \in$	**IdProc**			identificadores de proceso

Figura 8.1: Jauja básico: dominios semánticos

devolverá una continuación.

Habitualmente, el *estado* es el ente dinámico en el que se reflejan los cambios irreversibles producidos por la evaluación del programa; irreversibles por lo costoso que sería realizar, en tiempo de ejecución, una copia del estado completo para poder volver a versiones anteriores tras la salida de bloques. Por otro lado, el *entorno* se va definiendo de un modo local para cada bloque de programa, y sus referencias locales desaparecen cuando se sale de uno de ellos, por lo que el entorno es de menor tamaño que el estado, y su copia sí es factible. Esta distinción entre entorno y estado es habitual en los lenguajes imperativos, o en los que disponen de instrucciones imperativas —tipo la asignación a variables—. Sin embargo, en un lenguaje funcional, en el que cada variable se liga a un valor único, esta asociación no se rompe, por lo que con el adecuado renombramiento de variables de programa se puede disponer de un estado en el que no es necesario realizar la vuelta atrás. Nosotros nos encontramos en este segundo supuesto, por lo que optamos por fusionar entorno y estado, y tratarlo como el estado de los lenguajes imperativos.

Ahora bien, el estado necesario para evaluar denotacionalmente Jauja no consiste solamente en un entorno, **Env**, pues ya hemos insistido en nuestra intención de expresar en este modelo el paralelismo. Con este fin se incluye en el estado el *conjunto de canales*, **SChan**, en el que viene definido un grafo cuyos nodos serán los procesos creados en el estado y las aristas estarán etiquetadas con los valores comunicados por los canales.[2]

La definición del *entorno* es la habitual: liga identificadores, **Ide**, con valores en su sentido más amplio, o con el valor undefined, que indica que el identificador aún no ha sido usado. Por su parte, cada *canal*, **Chan**, del conjunto de canales, es una terna ⟨proceso productor, valor comunicado, proceso consumidor⟩.

[2]Tras observar el tipo de comunicaciones que por ahora tiene Jauja, se deduce que será un árbol.

Los *valores del entorno* incluyen los tres estados de evaluación de una variable:

1. Ya evaluada: está asociada al valor devuelto tras evaluar la expresión a la que estaba asociada en el programa.
2. Aún no ha sido demandada: se encuentra asociada a una clausura.
3. En proceso de evaluación: motivo por el que si es demandada de nuevo, la evaluación será errónea, y estaríamos ante un caso de *autorreferencia* como los descritos en la semántica operacional (Capítulo 6).

En el primer caso, el identificador estará ligado a un *valor de expresión*, **EVal**, o valor final, que en este subconjunto de Jauja solamente puede ser una *abstracción*, **Abs**. Sin embargo, la abstracción denotacional se acompaña de un conjunto de identificadores constituido por las variables libres de la abstracción sintáctica. Si bien este conjunto podría ser computado a partir del valor denotacional de la abstracción, sería de una manera más abstrusa que a partir de la abstracción sintáctica, motivo que nos lleva a incluirlo en los valores de expresión. Esta inclusión de variables libres viene motivada por el enfoque de "copia evaluada" (recordemos el Capítulo 6) que seguimos en este modelo denotacional.

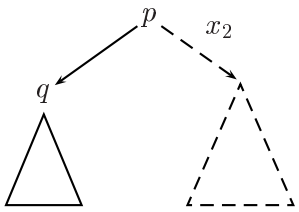
En el segundo caso, el de una *clausura*, **Clo**, ésta no estará aislada, sino que se le asocia un identificador de proceso, $\mathbf{IdProc} \times \mathbf{Clo}$. Dicho identificador indica qué proceso introdujo la variable en el entorno. El mencionado proceso será el padre, si la clausura da lugar a una creación de proceso. Sin embargo, esta información no es la que necesita la clausura para generar su propia descendencia, pues en este caso es necesario saber en qué proceso se evalúa dicha clausura. Este dato es un argumento que recibirá cada clausura al ser evaluada.

Ejemplo 8.1 *Relaciones entre identificadores de proceso y clausuras.*

Observemos el siguiente entorno:

$$\boxed{\begin{array}{l} x_1 \mapsto \lambda y.\nu_1 \\ x_2 \mapsto \langle p, \nu_2 \rangle \\ x_3 \mapsto \langle p, \mathcal{E}\,[\![x_1 \# x_2]\!]\,\rangle \end{array}}$$

Suponiendo que el objetivo sea evaluar x_3, observamos que dicha variable es del proceso p. Sin embargo, la clausura conlleva la creación de un nuevo proceso, al que nos referiremos como q, por lo que la aplicación $x_1\,x_2$ no será evaluada en el seno del proceso p sino que se hará en el de q. ¿Por qué es necesaria esta determinación de procesos? La respuesta es sencilla: si dicha aplicación conlleva la creación de otro proceso, éste último deberá ser hijo de q y nieto de p, y no hijo de p y hermano de q. Lo que sí que se evaluará "en su sitio", p, es x_2, y su resultado será enviado a q, por lo que las creaciones derivadas de la evaluación de x_2 constituirán otra línea sucesoria de p distinta de la formada por q y sus descendientes. Observemos una representación gráfica de este hecho:



Se distinguen en este árbol con línea continua aquellas líneas dinásticas que seguro se van a crear, mientras que se presentan con línea discontinua las potenciales descendencias.

□

Una clausura tiene significado en sí misma, una vez se le ha suministrado el identificador de proceso correspondiente. Sin embargo, y como ya se motivó en el Capítulo 7, necesitaremos enmarcarla en su contexto, para lo que será necesario suministrarle una continuación de expresión. De este modo, la clausura llega al punto de convertirse en una transformadora de estados.

Finalmente, el tercer caso de asociación de variable en un entorno es aquel que refleja el proceso de evaluación de dicha variable: la variable se asocia con el valor especial not_ready. Si tras aplicar una continuación inicial a un estado, en el estado final alguna variable está asociada a este valor not_ready, entonces queda detectada una autorreferencia —directa o indirecta— de variable.

En lo que concierne a los canales y los *valores de comunicación*, **CVal**, caben dos posibilidades:

1. Que el canal ya haya servido a su propósito de vía de transmisión.

2. Que el canal aún no haya sido utilizado.

En el primer caso, la terna del canal tendrá por valor un elemento de **EVal**, es decir, un valor propiamente dicho. En el segundo caso, la arista correspondiente al canal tendría por etiqueta el valor especial unsent.

8.1.2. Función de evaluación

Ya hemos explicado cómo el cómputo paralelo de Jauja hace necesario indicar, cuando se vaya a evaluar una expresión, el proceso en cuyo seno se llevará a cabo la evaluación, es decir, el padre de los posibles nuevos procesos. Nótese que este identificador de proceso no es necesario para distinguir en el entorno las variables de dicho proceso de las de otro, pues el modelo denotacional abstrae dicha información y las variables son copiadas virtualmente, es decir son compartidas en lugar de ser copiadas físicamente, como sucedía en las semánticas operacionales del Capítulo 6. Sin embargo, hemos visto que para decidir

quién es el padre de cada nuevo proceso es necesario este identificador de proceso. En consecuencia, la función semántica para evaluar expresiones tiene el siguiente tipo:

$$\mathcal{E} :: \mathbf{Exp} \to \mathbf{IdProc} \to \mathbf{ECont} \to \mathbf{Cont}.$$

Tras evaluar la expresión, la continuación de expresión contiene la información de qué hacer con el valor obtenido, y dado un estado, éste se transformará en un conjunto de estados, acumulando los efectos de evaluar la expresión y los de la continuación de expresión.

Los valores de la continuación de expresión inicial y el estado inicial vienen descritos en la siguiente definición:

Definición 8.1 Se definen la *continuación de expresión inicial* y el *estado inicial* de evaluación como:

$$\kappa_0 = id_\kappa = \lambda\varepsilon.\lambda s.\{s\},$$
$$s_0 = \langle \rho_0, \emptyset \rangle.$$

donde el *entorno inicial* ρ_0 contiene todos los identificadores ligados al valor especial undefined.

□

Si se desea localizar el valor correspondiente a la evaluación del programa E, se incluye en el entorno inicial la asociación $main \mapsto \langle main, \mathcal{E}\,[\![E]\!]\,\rangle$ y se evalúa la variable especial *main* partiendo de ese entorno.

La función semántica $\mathcal{E}$, que pasamos a explicar seguidamente, queda recogida en la Figura 8.2. Empleamos el operador $\oplus$ para extender/actualizar entornos, como por ejemplo en $\rho \oplus \{x \mapsto \langle p, \nu \rangle\}$ o en $s \oplus \{x \mapsto \langle p, \nu \rangle\}$, y $\oplus_{ch}$ en el caso de que se trate del conjunto de canales de un estado. Asimismo, las funciones newIde y newIdProc devuelven una variable y un identificador de proceso, respectivamente, frescos; la función card calcula el cardinal de un conjunto.

La evaluación de un *identificador* "fuerza" la evaluación del valor ligado a dicho identificador en el entorno suministrado. La función force se define en la Figura 8.3 y se explicará más adelante en la Sección 8.1.3.

Puede decirse que la evaluación de una λ-*abstracción* está finalizada; solamente hay que construir el valor denotacional que por naturaleza le corresponde: un par formado por una abstracción denotacional —función que dado un identificador devuelve una clausura— y el conjunto de variables libres de la abstracción sintáctica. Se aplica la continuación de expresión a este valor semántico.

Puesto que la *aplicación funcional* de Jauja es perezosa, la evaluación del argumento, x_2, tiene que ser postergada, y solamente se llevará a cabo si el argumento es demandado. Para conseguir modelizar esto, se sustituye la continuación de expresión κ por κ'. Esta nueva continuación de expresión será suministrada para la evaluación de la variable

$$\mathcal{E}\,[\![x]\!]\,p\,\kappa = \mathsf{force}\,x\,\kappa$$

$$\mathcal{E}\,[\![\backslash x.E]\!]\,p\,\kappa = \kappa\langle \lambda x.\mathcal{E}\,[\![E]\!]\,, \mathsf{fv}(\backslash x.E)\rangle$$

$$\begin{array}{l}\mathcal{E}\,[\![x_1\;x_2]\!]\,p\,\kappa = \mathcal{E}\,[\![x_1]\!]\,p\,\kappa' \\ \quad \texttt{donde}\ \kappa' = \lambda\langle\alpha, I\rangle.\lambda s.(\alpha\,x_2)\,p\,\kappa\,s\end{array}$$

$$\begin{array}{l}\mathcal{E}\,[\![x_1\#x_2]\!]\,p\,\kappa = \mathsf{forceFV}\,x_1\,\kappa' \\ \quad \texttt{donde}\ \kappa' = \lambda\langle\alpha, I\rangle.\lambda s.(\alpha\,i)\,q\,\kappa''\,(s \oplus \{i \mapsto \langle p, \mathcal{E}\,[\![x_2]\!]\,\rangle\}) \\ \qquad\quad q = \mathsf{newIdProc}\,s \\ \qquad\quad \langle\rho, \mathrm{sch}\rangle = s \\ \qquad\quad i = \mathsf{newIde}\,\rho \\ \qquad\quad \boxed{\begin{array}{l}\kappa''_{min} = \lambda\langle\alpha', I'\rangle.\lambda s'.\texttt{case}\ (\rho'\,i)\ \texttt{of} \\ \qquad \langle\alpha'', I''\rangle \in \mathbf{EVal} \longrightarrow \bigcup_{s_f \in S_f} \kappa\,\langle\alpha', I'\rangle\,s_f \\ \qquad\quad \texttt{donde}\ S_f = S_d \oplus_{ch} \{\langle q, \langle\alpha', I'\rangle, p\rangle\ , \langle p, \langle\alpha'', I''\rangle, q\rangle\} \\ \qquad \texttt{e.o.c.} \longrightarrow \bigcup_{s_f \in S_f} \kappa\,\langle\alpha', I'\rangle\,s_f \\ \qquad\quad \texttt{donde}\ S_f = S_c \oplus_{ch} \{\langle q, \langle\alpha, I\rangle, p\rangle\ , \langle p, \mathrm{unsent}, q\rangle\} \\ \qquad \texttt{endcase} \\ \quad \texttt{donde}\ \langle\rho', \mathrm{sch}'\rangle = s' \\ \qquad\quad S_c = \mathsf{mforceFV}\,I'\,s' \\ \qquad\quad S_d = \bigcup_{s_c \in S_c} \mathsf{mforceFV}\,I''\,s_c\end{array}} \\ \qquad\quad \boxed{\begin{array}{l}\kappa''_{max} = \lambda\langle\alpha', I'\rangle.\lambda s'.\ \bigcup_{s_f \in S_f} \kappa\,\langle\alpha', I'\rangle\,s_f \\ \quad \texttt{donde}\ S_c = \mathsf{mforceFV}\,I'\,s' \\ \qquad\quad S_f = \bigcup_{s_c \in S_c} \mathsf{forceFV}\,i\,\kappa_c\,s_c \\ \qquad\quad \kappa_c = \lambda\varepsilon''.\lambda s''.id_\kappa\,\varepsilon''\,(s'' \oplus_{ch} \{\langle q, \langle\alpha', I'\rangle, p\rangle, \langle p, \varepsilon'', q\rangle\})\end{array}}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,[\![\texttt{let}\ \{x_i = E_i\}_n\ \texttt{in}\ x]\!]\,p\,\kappa = \lambda\langle\rho, \mathrm{sch}\rangle.\mathcal{E}\,[\![x[y_1/x_1, \ldots, y_n/x_n]]\!]\,p\,\kappa'\,\langle\rho', \mathrm{sch}\rangle \\ \quad \texttt{donde}\ \{y_1, \ldots, y_n\} = \mathsf{newIde}\,n\,\rho \\ \qquad\quad \rho' = \rho \oplus \{y_i \mapsto \langle p, \mathcal{E}\,[\![E_i[y_1/x_1, \ldots, y_n/x_n]]\!]\,\rangle \mid 1 \le i \le n\} \\ \qquad\quad \boxed{\begin{array}{l}\kappa'_{min} = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa\,\varepsilon\,s' \\ \quad \texttt{donde}\ I = \{y_i \mid E_i \equiv x_1^i\#x_2^i \wedge (\rho\,y_i) \in \mathbf{IdProc} \times \mathbf{Clo}\} \\ \qquad\quad m = \mathsf{card}\,I \\ \qquad\quad \{q_1, \ldots, q_m\} = \mathsf{newIdProc}\,m\,\langle\rho, \mathrm{sch}\rangle \\ \qquad\quad s' = \langle\rho, \mathrm{sch}\rangle \oplus_{ch} \{\langle p, \mathrm{unsent}, q_j\rangle, \langle q_j, \mathrm{unsent}, p\rangle \mid 1 \le j \le m\}\end{array}} \\ \qquad\quad \boxed{\begin{array}{l}\kappa'_{max} = \lambda\varepsilon.\lambda s.\ \bigcup_{s_f \in S_f} \kappa\,\varepsilon\,s_f \\ \quad \texttt{donde}\ I = \{y_i \mid E_i \equiv x_1^i\#x_2^i \wedge 1 \le i \le n\} \\ \qquad\quad S_f = \mathsf{mforce}\,I\,s\end{array}}\end{array}$$

$$\mathcal{E}\,[\![x_1 \bowtie x_2]\!]\,p\,\kappa = \lambda s.((\mathcal{E}\,[\![x_1]\!]\,p\,\kappa\,s) \cup (\mathcal{E}\,[\![x_2]\!]\,p\,\kappa\,s))$$

Figura 8.2: Jauja básico: función de evaluación

que corresponde a la abstracción de la aplicación, x_1. Así, una vez que el valor de la abstracción haya sido obtenido, éste se aplica a la variable argumento, x_2, y se evalúa la correspondiente clausura, siendo κ la continuación de expresión que se pasa a dicha clausura.

El caso de la *creación de procesos*, o aplicación paralela, es más elaborado. La eva-

luación de $x_1 \# x_2$ provoca la creación de un proceso nuevo, q, teniendo lugar la siguiente secuencia de acciones:

1. Antes de crear el nuevo proceso se tiene que haber obtenido, a partir de x_1, el valor derivado de la correspondiente abstracción, $\langle \alpha, I \rangle$, y haber evaluado todas las variables libres pertinentes, I, —las dependencias libres que llamábamos en la semántica operacional (Capítulo 6)—. Estas evaluaciones se consiguen aplicando la función auxiliar forceFV —que se explicará en la Sección 8.1.3— a x_1. En el modelo operacional, todas estas variables libres ya evaluadas eran copiadas al *heap* del nuevo proceso, sin embargo, el modelo denotacional se abstrae de esta copia y comparte un único entorno para todos los procesos[3].

2. Se crea un nuevo proceso q, siendo q un nuevo identificador de proceso.

3. Se evalúa la clausura resultante de aplicar el valor $\langle \alpha, I \rangle$ —obtenido en el paso 1— al canal de entrada, i, habiéndole pasado como argumento el identificador q del nuevo proceso.

4. El valor obtenido en el paso 3, $\langle \alpha', I' \rangle$, tiene aún mucho porvenir. En su futuro tendrán poder de decisión tres continuaciones de expresión diferentes:

   a) κ: continuación de expresión de la creación de proceso,

   b) κ': continuación de expresión que lleva a cabo la aplicación, y

   c) κ'': continuación de expresión que refleja las comunicaciones en el estado final.

   Volviendo al valor $\langle \alpha', I' \rangle$, hay que aplicarle κ, y reflejar los cambios que esta aplicación produce en el estado s_f, resultante de llevar a cabo las comunicaciones; estas comunicaciones son realizadas gracias a κ''. Una creación de proceso implica dos comunicaciones:

   a) la que se deriva de la evaluación y uso del argumento/canal i, y

   b) la que comunica el resultado de la aplicación $(\alpha\, i)$.

   La segunda de ellas tiene que ser llevada a cabo siempre, para lo cual es necesario evaluar todas las variables libres asociadas al valor, I', obteniéndose el conjunto de estados S_c. La comunicación se refleja en los correspondientes conjuntos de canales, incorporando un canal de q a p cuyo valor es $\langle \alpha', I' \rangle$. Sin embargo, la primera de las comunicaciones dependerá, en muchas ocasiones, de la cota de especulación:

   Mínima (κ''_{min}): se analiza el estado para detectar si i ha sido demandado o no. En caso negativo no se comunicará, y el canal que se crea de p a q tomará el valor unsent; en caso afirmativo, se habrá obtenido el valor $\langle \alpha'', I'' \rangle$ y la comunicación tiene que proceder. No obstante, antes habrá que evaluar las variables

[3] Esto sólo es posible en el enfoque en el que las ligaduras se copian evaluadas, pues en el que se copian sin necesidad de estar evaluadas se pueden producir varias evaluaciones correspondientes a la misma variable de origen. Esto no tendría importancia de no ser por la presencia del no-determinismo: con "copia evaluada", en cada cómputo todos los procesos hijo dispondrán del mismo valor (obtenido de manera no-determinista) derivado de esa variable; con "copia sin evaluar", cada proceso hijo podría obtener un valor distinto para esa misma variable.

de I'' en cada estado de S_c, dando lugar al conjunto de estados S_d. La comunicación queda reflejada en el canal de p a q, que en este caso no contendrá el valor unsent sino el valor $\langle \alpha'', I'' \rangle$. El conjunto de estados resultante de aplicar la operación que corresponda es S_f.

Máxima (κ''_{max}): en este caso no es necesario detenerse a observar si i ha sido demandado o no, porque hay que desarrollar toda la evaluación especulativa posible. Por ello se fuerza la evaluación de las variables de I', dando lugar a S_c, y la evaluación del identificador (con extensión a dependencias libres) i en cada estado de este conjunto. Tras ello, se aplica la continuación de expresión κ_c al valor derivado del argumento, ε''. Esta continuación de expresión recoge los efectos de las comunicaciones, actualizando cada conjunto de canales, de manera que de p a q se ha comunicado $\langle \alpha', I' \rangle$, y desde q a p se ha comunicado ε''. El conjunto de estados tras estas operaciones es S_f.

Como resultado final de la continuación de expresión κ'', se aplica al valor de la aplicación, $\langle \alpha', I' \rangle$, la continuación de expresión de la aplicación paralela, κ, sobre cada uno de los estados de S_f.

En todo este proceso hemos tenido que introducir la variable i para asegurarnos de que en la semántica mínima solamente se produzcan las comunicaciones demandadas.[4]

Ejemplo 8.2 *Comunicación denotacional mínima.*

Consideremos el estado

$$\langle \{x_1 \mapsto \langle \lambda x.\mathcal{E}\,[\![x_2]\!]\,, \{x_2\}\rangle, x_2 \mapsto \langle p, 3+4\rangle, \}, \emptyset\rangle^5$$

y supongamos que tenemos que desarrollar

$$\mathcal{E}\,[\![x_1 \# x_2]\!]\,p\,id_\kappa$$

sobre el mismo. Observamos que x_2 se corresponde con el canal de entrada del proceso a crear, pero que no será demandada, por no aparecer x en el cuerpo de la abstracción denotacional. Sin embargo, x_2 es libre en dicho cuerpo, y su evaluación será forzada como paso necesario para la creación. ¿Quiere esto decir que el valor de x_2 tiene que ser comunicado a través del canal de entrada? No en la semántica mínima, en la que queremos que sólo se lleven a cabo las comunicaciones que han sido demandadas. La inclusión de la variable i en el proceso de obtención del valor denotacional evita esta comunicación.

□

Para evaluar una *declaración local de variables*, antes de evaluar su cuerpo hay que introducir en el entorno las variables declaradas. El modo de hacerlo, para evitar choques

[4] Para que las diferencias en las definiciones de κ''_{min} y κ''_{max} sean las menos posibles, esta variable también se usa en la semántica máxima.

[5] De nuevo los números y las operaciones aritméticas son usados como azúcar sintáctico de las correspondientes λ-abstracciones.

de nombres, requiere un renombramiento similar al realizado a tal efecto en la semántica operacional (Capítulo 6). Por cada x_i $(1 \leq i \leq n)$ se introduce y_i, una variable nueva que es ligada a la clausura $\mathcal{E}\,[\![E_i[y_j/x_j]]\!]$ $(1 \leq j \leq n)$. Cada una de estas clausuras es adornada con el identificador de proceso p, que es quien las introduce en el sistema y es su primer poseedor. La diferencia en relación al grado de especulación se expresa con la nueva continuación de expresión κ': en el caso de la semántica mínima, se tienen que crear los procesos de nivel superior, pero sin ejercer demanda sobre los canales, por lo que no tienen valor enviado, i.e. éste es unsent; en el de la máxima, se fuerzan todos los identificadores ligados a las clausuras que crean proceso —hecho detectado porque la expresión a la que estaba ligada la variable local correspondiente era una creación—. Al forzar este conjunto de variables se provoca que se evalúen todas las aplicaciones paralelas, con las correspondientes creaciones de canales, comunicaciones y forzado de variables descrito para la evaluación de creaciones de proceso.

Finalmente, la evaluación del *operador elección* comprende la posibilidad de evaluar de manera exclusiva una cualquiera de sus dos subexpresiones, x_1 y x_2.

8.1.3. Funciones semánticas auxiliares

En la definición de $\mathcal{E}$ se han empleado varias funciones auxiliares que se detallan en la Figura 8.3. Todas ellas se encargan de forzar la evaluación de determinados identificadores. No obstante, existen dos modos de proceder con las variables libres del valor obtenido; a saber, forzar su evaluación o dejarlas tal como se encuentran.

$$
\begin{array}{l}
\mathsf{force} :: \mathbf{Ide} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{force}\; x\; \kappa = \lambda\langle \rho, \mathrm{sch}\rangle.\mathtt{case}\ (\rho\; x)\ \mathtt{of} \\
\quad \varepsilon \in \mathbf{EVal} \longrightarrow \kappa\, \varepsilon\, \langle \rho, \mathrm{sch}\rangle \\
\quad \langle p, \nu\rangle \in (\mathbf{IdProc} \times \mathbf{Clo}) \longrightarrow \nu\, p\, \kappa'\, s' \\
\qquad \mathtt{donde}\ \kappa' = \lambda\varepsilon'.\lambda\langle \rho', \mathrm{sch}'\rangle.\kappa\, \varepsilon'\, \langle \rho' \oplus \{x \mapsto \varepsilon'\}, \mathrm{sch}'\rangle \\
\qquad\qquad s' = \langle \rho \oplus \{x \mapsto \mathrm{not\_ready}\}, \mathrm{sch}\rangle \\
\quad \mathsf{e.o.c.} \longrightarrow \mathsf{wrong}\, \langle \rho, \mathrm{sch}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{mforce} :: \mathbf{Ides} \to \mathbf{Cont} \\
\mathsf{mforce}\, \emptyset = \lambda s.\{s\} \\
\mathsf{mforce}\, (\{x\} \cup I) = \lambda s.\ \bigcup_{s' \in S'} \mathsf{mforce}\, I\, s' \\
\quad \mathtt{donde}\ S' = \mathsf{force}\, x\, id_\kappa\, s
\end{array}
$$

$$
\begin{array}{l}
\mathsf{forceFV} :: \mathbf{Ide} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{forceFV}\, x\, \kappa = \mathsf{force}\, x\, \kappa' \\
\quad \mathtt{donde}\ \kappa' = \lambda\langle \alpha, I\rangle.\lambda s'.\ \bigcup_{s'' \in S''} \kappa\, \langle \alpha, I\rangle\, s'' \\
\qquad\qquad S'' = \mathsf{mforceFV}\, I\, s'
\end{array}
$$

$$
\begin{array}{l}
\mathsf{mforceFV} :: \mathbf{Ides} \to \mathbf{Cont} \\
\mathsf{mforceFV}\, \emptyset = \lambda s.\{s\} \\
\mathsf{mforceFV}\, (\{x\} \cup I) = \lambda s.\ \bigcup_{s' \in S'} \mathsf{mforceFV}\, I\, s' \\
\quad \mathtt{donde}\ S' = \mathsf{forceFV}\, x\, id_\kappa\, s
\end{array}
$$

Figura 8.3: Jauja básico: funciones semánticas auxiliares

La función force compele la evaluación del identificador que se le pasa como primer argumento. Esta evaluación estará enmarcada en un contexto, recogido en la continuación de expresión que constituye su segundo argumento. Y, como sucede con la evaluación de una expresión en un contexto, se devuelve una continuación. A la hora de forzar una variable existen varias posibilidades:

- Que esté ligada a un valor de expresión: la única tarea que resta por realizar es aplicar el resto del cómputo —continuación de expresión κ— a dicho valor.

- Que esté ligada a una clausura: procede entonces evaluar dicha clausura en el proceso adecuado. La variable, mientras dura la evaluación, se encuentra ligada al valor not_ready. Cuando el valor haya sido obtenido, tendrá que ser ligado a la variable —asociación que efectúa la nueva continuación de expresión κ'— además de aplicarse la continuación de expresión inicial, κ, a dicho valor.

- Que se encuentre undefined o not_ready: ambos casos quedan recogidos en la última alternativa de la definición de force. El primer valor indica que la variable nunca ha sido declarada o definida; el segundo, se corresponde con la detección de una autorreferencia. En consecuencia, el intento de forzar la variable constituye un error, representado por la continuación wrong . Dicha continuación toma un estado y devuelve el conjunto unitario cuyo único elemento es dicho estado.

El cometido de mforce (multiple force) es forzar un conjunto de identificadores. Por ello, su definición recurre a la llamada reiterada de la función force. Puede observarse que el orden de las llamadas sería irrelevante, por lo que podemos fijar uno cualquiera, como de hecho hacemos. Ello es así por dos motivos:

- Si la evaluación de alguna de ellas no finaliza, la evaluación completa del programa quedaría indefinida.

- Si todas finalizan, la interferencia viene derivada de que una variable x_1 necesite para ser evaluada de la evaluación previa de otra, x_2. Se pueden dar dos situaciones:

  1. Que x_1 sea forzada primero, con lo que en el transcurso de su evaluación se tendrá que forzar la de x_2. Entonces, cuando llegue el turno de forzar x_2 dentro de la evaluación de mforce, ésta ya estará evaluada, por lo que no se modificará el estado.
  2. Que x_2 sea forzada en primer lugar. Posteriormente habrá que forzar x_1, y, cuando requiera del valor asociado a x_2, éste ya estará calculado.

  En ambas situaciones sucede realmente lo mismo, que el valor de x_2 es obtenido con anterioridad al de x_1.

En todos los casos, la continuación de expresión para cada uno de los identificadores es id_κ, que lo único que hace es devolver el conjunto de estados final.

La función de forzado extensivo a variables libres, forceFV (free variables force), invoca a force para evaluar el identificador inicial. Pero esta llamada se hace con una continuación de expresión nueva, que realiza dos acciones:

- Propagar el forzado extensivo a las dependencias libres de cada una de las variables libres, I, del valor devuelto, $\langle \alpha, I \rangle$.

- Aplicar la continuación de expresión κ a $\langle \alpha, I \rangle$, pero solamente en el ámbito de los estados de S'', resultado del forzado extensivo de identificadores de I.

El forzado múltiple extensivo a variables libres se realiza mediante mforceFV, función definida de manera similar a mforce, pero en este caso se invoca a forceFV.

Hasta aquí la definición de la semántica denotacional de continuaciones para el subconjunto de Jauja formado por el λ-cálculo básico con declaración local de variables y no-determinismo explícito, aunque simple.

En la siguiente sección veremos ejemplos ilustrativos de cómo se calcula el valor denotacional de una expresión.

8.1.4. Ejemplos

Veamos ahora algunas de las situaciones descritas en los ejemplos incluidos en el Capítulo 6 y otras nuevas de interés en el presente marco denotacional:

Ejemplo 8.3 *Agujero negro directo.*

En primer lugar vamos a considerar el caso propuesto en el Ejemplo 6.2:

$$\texttt{let } x_0 = x_0 \texttt{ in } x_0$$

Los valores iniciales son:

$$\begin{aligned}
&\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus \{main \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{let } x_0 = x_0 \texttt{ in } x_0]\!]\,\rangle\}\\
&\text{sch}_0 = \emptyset\\
&s_0 = \langle \rho_0, \text{sch}_0 \rangle\\
&\kappa_0 = id_\kappa\\
&p_0 = main
\end{aligned}$$

donde el identificador *main* es utilizado, al no haber lugar a confusión, tanto como el identificador del proceso principal, como el identificador principal de dicho proceso.

El cálculo del valor denotacional a partir de la expresión de *main* en el proceso principal se desarrolla como sigue:[6]

$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$

$\mathsf{force}\; main\,\kappa_0\,s_0 =$

$\mathcal{E}\,[\![\texttt{let } x_0 = x_0 \texttt{ in } x_0]\!]\,p_0\,\kappa_1\,s_1 =$

$\quad \kappa_1 = \lambda\varepsilon.\lambda\langle\rho, \text{sch}\rangle.\kappa_0\,\varepsilon\,\langle\rho \oplus \{main \mapsto \varepsilon\}, \text{sch}\rangle$

$\quad s_1 = \langle \rho_0 \oplus \{main \mapsto \text{not\_ready}\}, \text{sch}_0 \rangle$

$\mathcal{E}\,[\![x_1]\!]\,p_0\,\kappa_2\,s_2 =$

$\quad \kappa_2 = \kappa_1$ (mínima y máxima)

[6] Las indentaciones en los ejemplos indican cláusulas "donde", si bien no escribimos dicha palabra.

$s_2 = \langle \rho_1 \oplus \{x_1 \mapsto \langle p_0, \mathcal{E}\,[\![x_1]\!]\,\}, \mathrm{sch}_1\rangle$

$\mathsf{force}\; x_1\; \kappa_2\; s_2 =$

$\mathcal{E}\,[\![x_1]\!]\; p_0\; \kappa_3\; s_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_2\;\varepsilon\;\langle\rho \oplus \{x_1 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_3 = \langle \rho_2 \oplus \{x_1 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$

$\mathsf{force}\; x_1\; \kappa_3\; s_3 =$

$\mathsf{wrong}\; s_3 =$

$\{s_3\} =$

$\{\langle \rho_0 \oplus \{main \mapsto \mathrm{not\_ready}, x_1 \mapsto \mathrm{not\_ready}\}, \emptyset\rangle\}$

Se detectan tres hechos a partir del valor denotacional del programa:

1. Como la variable *main* no se encuentra ligada a ningún elemento de **EVal**, se deduce que la evaluación de la expresión inicial no da lugar a valor alguno. El valor especial not_ready no procede de una autorreferencia de la variable *main* a sí misma, ya que este caso es imposible por no existir más que una aparición de *main* en todo el entorno.

2. La variable x_1 se encuentra ligada al valor especial not_ready, de lo que se deduce que en su evaluación se ha producido una autorreferencia.

3. Dado que el conjunto de canales del sistema final es vacío, deducimos que no se ha desarrollado paralelismo, como era evidente a partir de la inexistencia de #-expresiones en el texto del programa.

En conclusión, la autorreferencia de x_1 ha conducido a la imposibilidad de obtener un valor de expresión para *main*.

□

Incluimos a continuación un ejemplo que muestra cómo se manifiesta la pereza del lenguaje en la evaluación de las variables locales de una declaración.

Ejemplo 8.4 *Pereza en la declaración local.*

Consideremos la expresión:

$$\texttt{let}\; x_1 = x_2\; x_3, x_2 = \backslash x.x, x_3 = 3\; \texttt{in}\; x_2.$$

El valor inicial del entorno es:[7]

$$\rho_0 = \{x_i \mapsto \mathrm{undefined}\} \oplus \{main \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{let}\; x_1 = x_2\; x_3, x_2 = \backslash x.x, x_3 = 3\; \texttt{in}\; x_2]\!]\,\rangle\}$$

El cálculo del valor denotacional a partir de *main* se desarrolla como sigue:

[7] El resto de valores iniciales son iguales a los del Ejemplo 8.3 y no serán repetidos ni en éste ni en los ejemplos que siguen.

$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$

$\mathsf{force}\;main\,\kappa_0\,s_0 =$

$\mathcal{E}\,[\![\texttt{let } x_1 = x_2\,x_3, x_2 = \backslash x.x, x_3 = 3 \texttt{ in } x_2]\!]\,p_0\,\kappa_1\,s_1 =$

$\quad \kappa_1 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_0\,\varepsilon\,\langle\rho \oplus \{main \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$\quad s_1 = \langle\rho_0 \oplus \{main \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_0\rangle$

$\mathcal{E}\,[\![x_5]\!]\,p_0\,\kappa_2\,s_2 =$

$\quad \kappa_2 = \kappa_1$

$\quad s_2 = \langle\rho_1 \oplus \{x_4 \mapsto \langle p_0, \mathcal{E}\,[\![x_5\,x_6]\!]\,\rangle, x_5 \mapsto \langle p_0, \mathcal{E}\,[\![\backslash x.x]\!]\,\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle\}, \mathrm{sch}_1\rangle$

$\mathsf{force}\;x_5\,\kappa_2\,s_2 =$

$\mathcal{E}\,[\![\backslash x.x]\!]\,p_0\,\kappa_3\,s_3 =$

$\quad \kappa_3 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_2\,\varepsilon\,\langle\rho \oplus \{x_5 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$\quad s_3 = \langle\rho_2 \oplus \{x_5 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$

$\kappa_3\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_3 =$

$\kappa_2\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,\langle\rho_2 \oplus \{x_5 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch}_2\rangle =$

$\kappa_1\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,\langle\rho_1 \oplus \{x_4 \mapsto \langle p_0, \mathcal{E}\,[\![x_5\,x_6]\!]\,\rangle, x_5 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle\}, \mathrm{sch}_1\rangle =$

$\kappa_0\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,\langle\rho_1 \oplus \{x_4 \mapsto \langle p_0, \mathcal{E}\,[\![x_5\,x_6]\!]\,\rangle, x_5 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle\}$
$\qquad \oplus\{main \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch}_1\rangle =$

$\{\langle\rho_0 \oplus \{x_4 \mapsto \langle p_0, \mathcal{E}\,[\![x_5\,x_6]\!]\,\rangle, x_5 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle, main \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \emptyset\rangle\}.$

A partir del valor denotacional del programa se observan varios hechos:

1. No se ha evaluado en paralelo; hecho que queda patente en el conjunto de canales vacío del sistema final.
2. La variable *main* se encuentra ligada a un elemento de **EVal**, $\langle 3, \emptyset\rangle$.
3. De las restantes variables, x_5 —que proviene de la inicial x_2— se encuentra también ligada a un valor de **EVal**, por lo que se deduce que ha sido demandada para la evaluación de *main*; no cabe otra posibilidad al no haberse desarrollado ningún paralelismo.
4. El resto de variables locales permanecen asociadas a elementos de $\mathbf{IdProc} \times \mathbf{Clo}$, lo que permite deducir que no han sido demandadas para la evaluación de la variable principal.

En suma, en el modelo denotacional expuesto queda patente la pereza en la evaluación de la declaración local de variables: basta observar a qué se encuentran ligadas las distintas variables del entorno.

□

En el siguiente ejemplo se tiene que crear un proceso que, al necesitarse su valor para construir el valor de la variable *main*, i.e. al tratarse de un proceso especulativo,

da lugar al mismo valor denotacional, tanto en el caso mínimo como en el máximo.

Ejemplo 8.5 *Paralelismo no especulativo.*

Consideremos la expresión:

$$\texttt{let } x_1 = x_2\#x_3, x_2 = \backslash x.x, x_3 = 3 \texttt{ in } x_1.$$

El entorno inicial es:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus \{main \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{let } x_1 = x_2\#x_3, x_2 = \backslash x.x, x_3 = 3 \texttt{ in } x_1]\!]\,\rangle\}$$

El cálculo del valor denotacional a partir de *main* se desarrolla como sigue:

$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$

$\mathsf{force}\,main\,\kappa_0\,s_0 =$

$\mathcal{E}\,[\![\texttt{let } x_1 = x_2\#x_3, x_2 = \backslash x.x, x_3 = 3 \texttt{ in } x_1]\!]\,p_0\,\kappa_1\,s_1 =$

$\kappa_1 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sdh}\rangle.\kappa_0\,\varepsilon\,\langle\rho \oplus \{main \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_1 = \langle\rho_0 \oplus \{main \mapsto \text{not\_ready}\}, \mathrm{sdh}_0\rangle$

$\mathcal{E}\,[\![x_4]\!]\,p_0\,\kappa_2\,s_2 =$[8]

mínima	máxima
$\kappa_2 = \kappa_1$ (iguales porque la creación de x_4 es demandada)	$\kappa_2 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sdh}\rangle.\bigcup_{s_f \in S_f} \kappa_1\,\varepsilon\,s_f$ $S_f = \mathsf{mforce}\,\{x_4\}\,\langle\rho, \mathrm{sdh}\rangle$

$s_2 = \langle\rho_1 \oplus \{x_4 \mapsto \langle p_0, \mathcal{E}\,[\![x_5\#x_6]\!]\,\rangle, x_5 \mapsto \langle p_0, \mathcal{E}\,[\![\backslash x.x]\!]\,\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle\}, \mathrm{sdh}_1\rangle$

$\mathsf{force}\,x_4\,\kappa_2\,s_2 =$

$\mathcal{E}\,[\![x_5\#x_6]\!]\,p_0\,\kappa_3\,s_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sdh}\rangle.\kappa_2\,\varepsilon\,\langle\rho \oplus \{x_4 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_3 = \langle\rho_2 \oplus \{x_4 \mapsto \text{not\_ready}\}, \mathrm{sdh}_2\rangle$

$\mathsf{forceFV}\,x_5\,\kappa_4\,s_3 =$

$\kappa_4 = \lambda\langle\alpha, I\rangle.\lambda s.(\alpha\,i)\,p_1\,\kappa_5\,(s \oplus \{i \mapsto \langle p_0, \mathcal{E}\,[\![x_6]\!]\,\rangle\})$

mínima	máxima
$\kappa_5 = \lambda\langle\alpha', I'\rangle.\lambda s'.\bigcup_{s_f \in S_f} \kappa_3\,\langle\alpha', I'\rangle\,s_f$ $S_f = S_d \oplus_{ch} \{\langle p_1, \langle\alpha', I'\rangle, p_0\rangle, \langle p_0, \langle\alpha'', I''\rangle, p_1\rangle\}$ $\langle\alpha'', I''\rangle = \rho'\,i$ $\langle\rho', \mathrm{sdh}'\rangle = s'$ $S_c = \mathsf{mforceFV}\,I'\,s'$ $S_d = \bigcup_{s_c \in S_c} \mathsf{mforceFV}\,I''\,s_c$	$\kappa_5 = \lambda\langle\alpha', I'\rangle.\lambda s'.\bigcup_{s_f \in S_f} \kappa_3\,\langle\alpha', I'\rangle\,s_f$ $S_c = \mathsf{mforceFV}\,I'\,s'$ $S_f = \bigcup_{s_c \in S_c} \mathsf{forceFV}\,i\,\kappa_c\,s_c$ $\kappa_c = \lambda\varepsilon''.\lambda s''.id_\kappa\,\varepsilon''\,s'''$ $s''' = s'' \oplus_{ch} \{\langle p_1, \langle\alpha', I'\rangle, p_0\rangle, \langle p_0, \varepsilon'', p_1\rangle\}$

[8] En casos como estos la continuación de expresión y, posiblemente, otros parámetros dependen de si se trata de la semántica mínima o de la máxima.

$\mathsf{force}\, x_5\, \kappa_6\, s_3 =$

$\kappa_6 = \lambda\langle \alpha, I\rangle.\lambda s'. \bigcup_{s'' \in S''} \kappa_4\, \langle \alpha, I\rangle\, s''$

$S'' = \mathsf{mforceFV}\, I\, s'$

$\mathcal{E}\, [\![\backslash x.x]\!]\, p_0\, \kappa_7\, s_5 =$

$\kappa_7 = \lambda\varepsilon.\lambda\langle \rho, \mathrm{sdh}\rangle.\kappa_6\, \varepsilon\, \langle \rho \oplus \{x_5 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_4 = \langle \rho_3 \oplus \{x_5 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_3\rangle$

$\kappa_7\, \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle\, s_4 =$

$\kappa_6\, \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle\, s_5 =$

$s_5 = \langle \rho_4 \oplus \{x_5 \mapsto \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle\}, \mathrm{sdh}_4\rangle$

$\bigcup_{s_{mf} \in \mathsf{mforceFV}\, \emptyset\, s_5} \kappa_4\, \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle\, s_{mf} =$

$\kappa_4\, \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle\, s_5 =$

$\mathcal{E}\, [\![i]\!]\, p_1\, \kappa_5\, s_6 =$

$s_6 = s_5 \oplus \{i \mapsto \langle p_0, \mathcal{E}\, [\![x_6]\!]\, \rangle\}$

$\mathsf{force}\, i\, \kappa_5\, s_6 =$

$\mathcal{E}\, [\![x_6]\!]\, p_0\, \kappa_8\, s_7 =$ (Obsérvese que el valor de x_6 tiene que obtenerlo p_0).

$\kappa_8 = \lambda\varepsilon.\lambda\langle \rho, \mathrm{sdh}\rangle.\kappa_5\, \varepsilon\, \langle \rho \oplus \{i \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_7 = \langle \rho_6 \oplus \{i \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_6\rangle$

$\mathsf{force}\, x_6\, \kappa_8\, s_7 =$

$\mathcal{E}\, [\![3]\!]\, p_0\, \kappa_9\, s_8 =$

$\kappa_9 = \lambda\varepsilon.\lambda\langle \rho, \mathrm{sdh}\rangle.\kappa_8\, \varepsilon\, \langle \rho \oplus \{x_6 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_8 = \langle \rho_7 \oplus \{x_6 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_7\rangle$

$\kappa_9\, \langle 3, \emptyset\rangle\, s_8 =$

$\kappa_8\, \langle 3, \emptyset\rangle\, \langle \rho_8 \oplus \{x_6 \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sdh}_8\rangle =$

$\kappa_5\, \langle 3, \emptyset\rangle\, \langle \rho_5 \oplus \{x_6 \mapsto \langle 3, \emptyset\rangle, i \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sdh}_5\rangle =$

a) Según κ_5 de la semántica mínima

$\bigcup_{s_f \in S_f} \kappa_3\, \langle 3, \emptyset\rangle\, s_f =$

$S_f = S_d \oplus_{ch} \{\langle p_1, \langle 3, \emptyset\rangle, p_0\rangle, \langle p_0, \langle 3, \emptyset\rangle, p_1\rangle\}$

$S_c = \{\langle \rho_5 \oplus \{x_6 \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sdh}_5\rangle\}$

$S_d = S_c$

$\kappa_3\, \langle 3, \emptyset\rangle\, \langle \rho_4 \oplus \{x_5 \mapsto \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle, x_6 \mapsto \langle 3, \emptyset\rangle, i \mapsto \langle 3, \emptyset\rangle\}, \{\langle p_1, \langle 3, \emptyset\rangle, p_0\rangle, \langle p_0, \langle 3, \emptyset\rangle, p_1\rangle\}\rangle =$

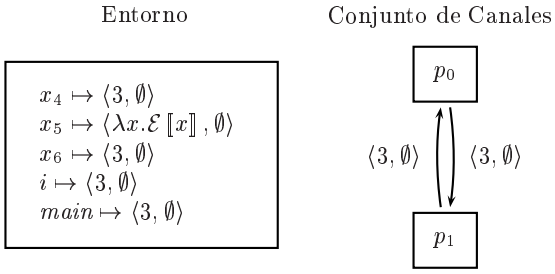
$\kappa_2\, \langle 3, \emptyset\rangle\, \langle \rho_3 \oplus \{x_5 \mapsto \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle, x_6 \mapsto \langle 3, \emptyset\rangle, i \mapsto \langle 3, \emptyset\rangle, x_4 \mapsto \langle 3, \emptyset\rangle\}, \{\langle p_1, \langle 3, \emptyset\rangle, p_0\rangle, \langle p_0, \langle 3, \emptyset\rangle, p_1\rangle\}\rangle =$

$\kappa_1\, \langle 3, \emptyset\rangle\, \langle \rho_2 \oplus \{x_4 \mapsto \langle 3, \emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\, [\![x]\!]\, , \emptyset\rangle, x_6 \mapsto \langle 3, \emptyset\rangle, i \mapsto \langle 3, \emptyset\rangle\}, \{\langle p_1, \langle 3, \emptyset\rangle, p_0\rangle, \langle p_0, \langle 3, \emptyset\rangle, p_1\rangle\}\rangle =$

$$\kappa_0\,\langle 3,\emptyset\rangle\,\langle \rho_1 \oplus \{x_4 \mapsto \langle 3,\emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,,\emptyset\rangle, x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle, main \mapsto \langle 3,\emptyset\rangle\},$$
$$\{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle =$$

$$\{\langle \rho_0 \oplus \{x_4 \mapsto \langle 3,\emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,,\emptyset\rangle, x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle, main \mapsto \langle 3,\emptyset\rangle\},$$
$$\{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle\}.$$

El estado final correspondiente a la semántica mínima es:



b) Según κ_5 de la semántica máxima

$$\bigcup_{s_f \in S_f} \kappa_3\,\langle 3,\emptyset\rangle\,s_f =$$

$$S_c = \mathsf{mforceFV}\,\emptyset\,s_9 = \{s_9\}$$

$$s_9 = \langle \rho_5 \oplus \{x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle\}, \mathrm{sch}_5\rangle$$

$$S_f = \bigcup_{s_c \in S_c} \mathsf{forceFV}\,i\,\kappa_c\,s_c =$$

$$\mathsf{forceFV}\,i\,\kappa_c\,s_9 =$$

$$\mathsf{force}\,i\,\kappa_{10}\,s_9 =$$

$$\kappa_{10} = \lambda\langle\alpha, I\rangle.\lambda s'.\bigcup_{s'' \in S''} \kappa_c\,\langle\alpha, I\rangle\,s''$$

$$S'' = \mathsf{mforceFV}\,I\,s'$$

$$\kappa_{10}\,\langle 3,\emptyset\rangle\,s_9 =$$

$$\bigcup_{s_{mf} \in S_{mf}} \kappa_c\,\langle 3,\emptyset\rangle\,s_{mf} =$$

$$S_{mf} = \mathsf{mforceFV}\,\emptyset\,s_9 = \{s_9\}$$

$$\kappa_c\,\langle 3,\emptyset\rangle\,s_9 =$$

$$id_\kappa\,\langle 3,\emptyset\rangle\,(s_9 \oplus_{ch} \{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\rangle 3,\emptyset\rangle,p_1\rangle\}) =$$

$$\{\langle \rho_5 \oplus \{x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle\}, \mathrm{sch}_5 \oplus \{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle\}$$

$$\kappa_3\,\langle 3,\emptyset\rangle\,\langle \rho_5 \oplus \{x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle\}, \mathrm{sch}_5 \oplus \{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle =$$

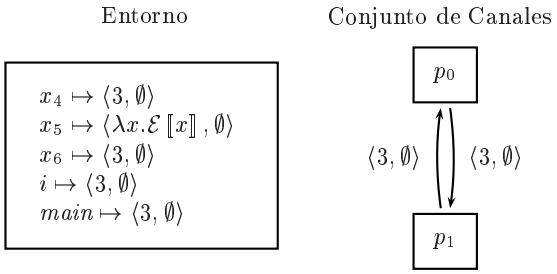
$$\kappa_2\,\langle 3,\emptyset\rangle\,\langle \rho_4 \oplus \{x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,,\emptyset\rangle, x_4 \mapsto \langle 3,\emptyset\rangle\},$$
$$\mathrm{sch}_4 \oplus \{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle =$$

$$\kappa_1\,\langle 3,\emptyset\rangle\,\langle \rho_3 \oplus \{x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,,\emptyset\rangle, x_4 \mapsto \langle 3,\emptyset\rangle\},$$
$$\mathrm{sch}_4\{\oplus\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle =$$

$$\kappa_0\,\langle 3,\emptyset\rangle\,\langle \rho_2 \oplus \{x_4 \mapsto \langle 3,\emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,,\emptyset\rangle, x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle, main \mapsto \langle 3,\emptyset\rangle\},$$
$$\{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle =$$

$$\{\langle \rho_0 \oplus \{x_4 \mapsto \langle 3,\emptyset\rangle, x_5 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,,\emptyset\rangle, x_6 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle, main \mapsto \langle 3,\emptyset\rangle\},$$
$$\{\langle p_1,\langle 3,\emptyset\rangle,p_0\rangle,\langle p_0,\langle 3,\emptyset\rangle,p_1\rangle\}\rangle\}.$$

El estado final correspondiente a la semántica máxima es:



Podemos observar que los valores denotacionales del programa para las semánticas mínima y máxima coinciden exactamente. Nótese que en el modelo denotacional se conserva la topología de procesos que han sido creados, ello vía el conjunto de canales de comunicación; en el caso de la semántica operacional vista en el Capítulo 6 esto no era posible observando el sistema final, pues las conexiones desaparecían del sistema una vez que habían cumplido su cometido.

En conclusión, hemos podido observar que la semántica mínima y la máxima no difieren cuando no hay paralelismo especulativo.

□

En cambio, en el siguiente ejemplo se presenta un caso de explotación del paralelismo especulativo:

Ejemplo 8.6 *Paralelismo especulativo.*

Estudiemos la semántica de la expresión siguiente:

$$\texttt{let}\ x_1 = x_2\#x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2\ x_3\ \texttt{in}\ x_2$$

El entorno inicial es:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus \{main \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{let}\ x_1 = x_2\#x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2\ x_3\ \texttt{in}\ x_2]\!]\,\rangle\}$$

Pasemos a calcular el valor denotacional correspondiente a esta expresión:

$$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$$

$$\mathsf{force}\,main\,\kappa_0\,s_0 =$$

$$\mathcal{E}\,[\![\texttt{let}\ x_1 = x_2\#x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2\ x_3\ \texttt{in}\ x_2]\!]\,p_0\,\kappa_1\,s_1 =$$

$\kappa_1 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_0\ \varepsilon\ \langle\rho \oplus \{main \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_1 = \langle\rho_0 \oplus \{main \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_0\rangle$

$\mathcal{E}\,[\![x_6]\!]\ p_0\ \kappa_2\ s_2 =$

mínima	máxima
$\kappa_2 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_1\ \varepsilon\ s'$	$\kappa_2 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\ \bigcup_{s_f \in S_f} \kappa_1\ \varepsilon\ s_f$
$I = \{x_5\}$	$S_f = \mathsf{mforce}\ \{x_5\}\ \langle\rho, \mathrm{sch}\rangle$
$m = 1$	
$\{q_1, \ldots, q_m\} = \mathsf{newIdProc}\ m\ \langle\rho, \mathrm{sch}\rangle$	
$s' = \langle\rho, \mathrm{sch}\rangle \oplus_{ch} \{\langle p_0, unsent, q_j\rangle, \langle q_j, unsent, p_0\rangle \mid 1 \leq j \leq m\}$	

$s_2 = \langle\rho_1 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![x_6 \# x_7]\!]\,\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![\backslash x.x]\!]\,\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![x_6\ x_7]\!]\,\rangle\}, \mathrm{sch}_1\rangle$

$\mathsf{force}\ x_6\ \kappa_2\ s_2 =$

$\mathcal{E}\,[\![\backslash x.x]\!]\ p_0\ \kappa_3\ s_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_2\ \varepsilon\ \langle\rho \oplus \{x_6 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_3 = \langle\rho_2 \oplus \{x_6 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$

$\kappa_3\ \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\ s_3 =$

$\kappa_2\ \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\ \langle\rho_2 \oplus \{x_6 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch}_2\rangle =$

a) Para la semántica mínima tenemos

$\kappa_1\ \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\ s_4 =$

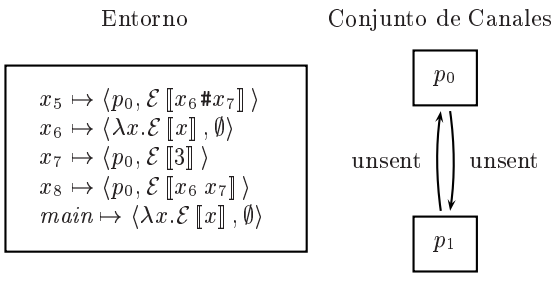
$s_4 = \langle\rho_1 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![x_6 \# x_7]\!]\,\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![x_6\ x_7]\!]\,\rangle, x_6 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\},$
$\mathrm{sch}_1 \oplus \{\langle p_0, \mathrm{unsent}, p_1\rangle, \langle p_1, \mathrm{unsent}, p_0\rangle\}\rangle$

$\kappa_0\ \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\ s_5 =$

$s_5 = \langle\rho_4 \oplus \{main \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch}_4\rangle$

$\{\langle\rho_0 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![x_6 \# x_7]\!]\,\rangle, x_6 \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle,$
$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![x_6\ x_7]\!]\,\rangle, main \mapsto \langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\},$
$\{\langle p_0, \mathrm{unsent}, p_1\rangle, \langle p_1, \mathrm{unsent}, p_0\rangle\}\rangle\}.$

El estado final de la semántica mínima es:



b) Para la semántica máxima tenemos

$$\bigcup_{s_f \in S_f} \kappa_1 \langle \lambda x.\mathcal{E}[\![x]\!], \emptyset\rangle\, s_f = (*)$$

$$S_f = \mathsf{mforce}\,\{x_5\}\, s_6 =$$

$$s_6 = \langle \rho_2 \oplus \{x_6 \mapsto \langle \lambda x.\mathcal{E}[\![x]\!], \emptyset\rangle\}, \mathrm{sch}_2\rangle$$

$$\bigcup_{s' \in S'} \mathsf{mforce}\,\emptyset\, s' = \bigcup_{s' \in S'} \{s'\} = S'$$

$$S' = \mathsf{force}\, x_5\, id_\kappa\, s_6 =$$

$$\mathcal{E}[\![x_6 \# x_7]\!]\, p_0\, \kappa_4\, s_7 =$$

$$\kappa_4 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.id_\kappa\, \varepsilon\, \langle \rho \oplus \{x_5 \mapsto \varepsilon\}, \mathrm{sch}\rangle$$

$$s_7 = \langle \rho_6 \oplus \{x_5 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_6\rangle$$

$$\mathsf{forceFV}\, x_6\, \kappa_5\, s_7 =$$

$$\kappa_5 = \lambda\langle\alpha, I\rangle.\lambda s.(\alpha\, i)\, p_1\, \kappa_6\, (s \oplus \{i \mapsto \langle p_0, \mathcal{E}[\![x_7]\!]\rangle\})$$

$$\kappa_6 = \lambda\langle\alpha', I'\rangle.\lambda s'.\bigcup_{s_f \in S_f} \kappa_4\, \langle \alpha', I'\rangle\, s_f$$

$$S_c = \mathsf{mforceFV}\, I'\, s'$$

$$S_f = \bigcup_{s_c \in S_c} \mathsf{forceFV}\, i\, \kappa_c\, s_c$$

$$\kappa_c = \lambda\varepsilon''.\lambda s''.id_\kappa\, \varepsilon''\, (s'' \oplus_{ch} \{\langle p_1, \langle \alpha', I'\rangle, p_0\rangle, \langle p_0, \varepsilon'', p_1\rangle\})$$

$$\mathsf{force}\, x_6\, \kappa_7\, s_7 =$$

$$\kappa_7 = \lambda\langle\alpha, I\rangle.\lambda s'.\bigcup_{s'' \in S''} \kappa_5\, \langle \alpha, I\rangle\, s''$$

$$S'' = \mathsf{mforceFV}\, I\, s'$$

$$\kappa_7\, \langle \lambda x.\mathcal{E}[\![x]\!], \emptyset\rangle\, s_7 =$$

$$\bigcup_{s'' \in S''} \kappa_5\, \langle \lambda x.\mathcal{E}[\![x]\!], \emptyset\rangle\, s'' =$$

$$S'' = \mathsf{mforceFV}\, \emptyset\, s_7 = \{s_7\}$$

$$\kappa_5\, \langle \lambda x.\mathcal{E}[\![x]\!], \emptyset\rangle\, s_7 =$$

$$\mathcal{E}[\![i]\!]\, p_1\, \kappa_6\, s_8 =$$

$$s_8 = s_7 \oplus \{i \mapsto \langle p_0, \mathcal{E}[\![x_7]\!]\rangle\}$$

$$\mathsf{force}\, i\, \kappa_6\, s_8 =$$

$$\mathcal{E}[\![x_7]\!]\, p_1\, \kappa_8\, s_9 =$$

$$\kappa_8 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_6\, \varepsilon\, \langle \rho \oplus \{i \mapsto \varepsilon\}, \mathrm{sch}\rangle$$

$$s_9 = \langle \rho_8 \oplus \{i \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_8\rangle$$

$$\mathsf{force}\, x_7\, \kappa_8\, s_9 =$$

$$\mathcal{E}[\![3]\!]\, p_0\, \kappa_9\, s_{10} =$$

$$\kappa_9 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_8\, \varepsilon\, \langle \rho \oplus \{x_7 \mapsto \varepsilon\}, \mathrm{sch}\rangle$$

$$s_{10} = \langle \rho_9 \oplus \{x_7 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_9\rangle$$

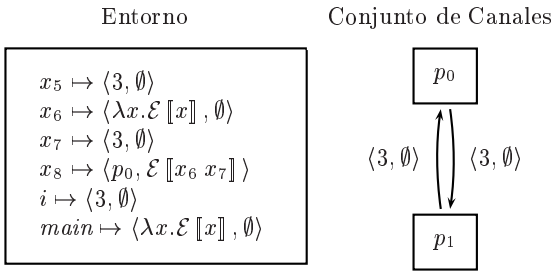
$$\kappa_9\, \langle 3, \emptyset\rangle\, s_{10} =$$

$$\kappa_8\, \langle 3, \emptyset\rangle\, s_{11} =$$

$$s_{11} = \langle \rho_{10} \oplus \{x_7 \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sch}_{10}\rangle$$

$$\kappa_6\,\langle 3,\emptyset\rangle\,s_{12} =$$
$$\quad s_{12} = \langle \rho_{11} \oplus \{i \mapsto \langle 3,\emptyset\rangle\}, \mathrm{sch}_{11}\rangle$$
$$\bigcup_{s_f \in S_f} \kappa_4\,\langle 3,\emptyset\rangle\,s_f =$$
$$\quad S_c = \mathsf{mforceFV}\,\emptyset\,s_{12} = \{s_{12}\}$$
$$\quad S_f = \mathsf{forceFV}\,i\,\kappa_c\,s_{12} =$$
$$\quad\quad \mathsf{force}\,i\,\kappa_{10}\,s_{12} =$$
$$\quad\quad\quad \kappa_{10} = \lambda\langle\alpha, I\rangle.\lambda s'. \bigcup_{s'' \in S''} \kappa_c\,\langle\alpha, I\rangle\,s''$$
$$\quad\quad\quad S'' = \mathsf{mforceFV}\,I\,s'$$
$$\quad\quad \kappa_{10}\,\langle 3,\emptyset\rangle\,s_{12} =$$
$$\quad\quad \kappa_c\,\langle 3,\emptyset\rangle\,s_{12} =$$
$$\quad\quad id_\kappa\,\langle 3,\emptyset\rangle\,(s_{12} \oplus_{ch} \{\langle p_1, \langle\langle 3,\emptyset\rangle\rangle, p_0\rangle, \langle p_0, \langle\langle 3,\emptyset\rangle\rangle, p_1\rangle\}) =$$
$$\quad\quad \{s_{13}\}$$
$$\quad\quad\quad s_{13} = \langle \rho_7 \oplus \{x_7 \mapsto \langle 3,\emptyset\rangle, i \mapsto \langle 3,\emptyset\rangle\}, \mathrm{sch}_0 \oplus \langle p_1, \langle\langle 3,\emptyset\rangle\rangle, p_0\rangle, \langle p_0, \langle\langle 3,\emptyset\rangle\rangle, p_1\rangle\}\rangle$$
$$\kappa_4\,\langle 3,\emptyset\rangle\,s_{13} =$$
$$id_\kappa\,\langle 3,\emptyset\rangle\,s_{14} =$$
$$\quad s_{14} = \langle \rho_{13} \oplus \{x_5 \mapsto \langle 3,\emptyset\rangle\}, \mathrm{sch}_{13}\rangle$$
$$\{s_{14}\} = S' = S_f$$
$$(*) = \kappa_1\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_{14} =$$
$$\kappa_0\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,\langle \rho_{14} \oplus \{main \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch}_{14}\rangle =$$
$$\{\langle \rho_0 \oplus \{x_5 \mapsto \langle 3,\emptyset\rangle, x_6 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \langle 3,\emptyset\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![x_6\,x_7]\!]\,\rangle, i \mapsto \langle 3,\emptyset\rangle, main \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\},$$
$$\quad \{\langle p_1, \langle 3,\emptyset\rangle, p_0\rangle \langle p_0, \langle 3,\emptyset\rangle, p_1\rangle\})\}.$$

El estado final de la semántica máxima es:



Observando los valores denotacionales que hemos obtenido se puede concluir:

1. En la semántica mínima el nuevo proceso que se ha creado no ha producido ningún valor, en tanto que en la máxima sí, de lo que deducimos que en la semántica máxima este proceso era especulativo.

2. En ambos casos la variable *main* se encuentra ligada a un elemento de **EVal**, $\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle$.

3. De las restantes variables, en la semántica máxima todas salvo x_8 se encuentran también ligadas a un valor de **EVal**, con lo que se deduce que han sido demandadas para la evaluación de la variable *main* o del proceso especulativo. Pero si observamos las variables x_5 y x_7, en el sistema de la semántica mínima vemos que se asocian a clausuras, de lo que deducimos que no han sido necesarias para la obtención del valor principal y sí para calcular alguno de los valores relacionados con el proceso especulativo.

4. El hecho de que en la semántica mínima x_8 no se encuentre asociada a un valor de **EVal**, indica que no ha sido demandada para la obtención del valor de la variable principal. Tampoco en la semántica máxima se encuentra ligada a un valor de **EVal**. Se observa que el paralelismo que se explota en Jauja no es todo el implícito en el programa, sino solamente aquel indicado con `#`-expresiones.

En suma, la semántica denotacional permite observar la explotación de paralelismo especulativo, y determinar qué procesos han sido especulativos, comparando la semántica máxima y la mínima. Y más allá de tal determinación también podemos precisar las variables que han sido evaluadas por causa de la especulación.

□

Veamos finalmente un ejemplo en el que se puede apreciar la explosión en el número de estados cuando se introduce no-determinismo.

Ejemplo 8.7 *No-determinismo.*

Estudiemos la semántica de la expresión:

$$\texttt{let } x_1 = x_2 \bowtie x_3, x_2 = x_4 \bowtie x_5, x_3 = 3, x_4 = 4, x_5 = 5 \texttt{ in } x_1$$

El entorno inicial es:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus \{main \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{let } x_1 = x_2 \bowtie x_3, x_2 = x_4 \bowtie x_5, x_3 = 3, x_4 = 4, x_5 = 5 \texttt{ in } x_1]\!]\,\rangle\}$$

Pasemos a calcular el valor denotacional correspondiente a esta expresión:

$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$

$\mathsf{force}\,main\,\kappa_0\,s_0 =$

$\mathcal{E}\,[\![\texttt{let } x_1 = x_2 \bowtie x_3, x_2 = x_4 \bowtie x_5, x_3 = 3, x_4 = 4, x_5 = 5 \texttt{ in } x_1]\!]\,p_0\,\kappa_1\,s_1 =$

$\kappa_1 = \lambda\varepsilon.\lambda\langle\rho, \text{sch}\rangle.\kappa_0\,\varepsilon\,\langle\rho \oplus \{main \mapsto \varepsilon\}, \text{sch}\rangle$

$s_1 = \langle\rho_0 \oplus \{main \mapsto \text{not\_ready}\}, \text{sch}_0\rangle$

$\mathcal{E}\,[\![x_6]\!]\,p_0\,\kappa_2\,s_2 =$

$\kappa_2 = \kappa_1$ (mínima o máxima)

$s_2 = \langle\rho_1 \oplus \{x_6 \mapsto \langle p_0, \mathcal{E}\,[\![x_7 \bowtie x_8]\!]\,\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![x_9 \bowtie x_{10}]\!]\,\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle,$

$$x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle, x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle\}, \mathrm{sch}_1\rangle$$

$\mathsf{force}\, x_6\, \kappa_2\, s_2 =$

$\mathcal{E}\,[\![x_7 \bowtie x_8]\!]\,\mathcal{P}_0\,\kappa_3\,s_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_2\,\varepsilon\,\langle\rho \oplus \{x_6 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_3 = \langle\rho_2 \oplus \{x_6 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$

$(\mathcal{E}\,[\![x_7]\!]\,p_0\,\kappa_3\,s_3) \cup (\mathcal{E}\,[\![x_8]\!]\,p_0\,\kappa_3\,s_3) =$

En este punto la evaluación se divide en dos líneas:

(1) $\boxed{\mathcal{E}\,[\![x_7]\!]\,p_0\,\kappa_3\,s_3 =}$

$\mathsf{force}\, x_7\, \kappa_3\, s_3 =$

$\mathcal{E}\,[\![x_9 \bowtie x_{10}]\!]\,p_0\,\kappa_4\,s_4 =$

$\kappa_4 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_3\,\varepsilon\,\langle\rho \oplus \{x_7 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_4 = \langle\rho_3 \oplus \{x_7 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_3\rangle$

$(\mathcal{E}\,[\![x_9]\!]\,p_0\,\kappa_4\,s_4) \cup (\mathcal{E}\,[\![x_{10}]\!]\,p_0\,\kappa_4\,s_4) =$

De nuevo la línea se divide en dos:

(1.1) $\boxed{\mathcal{E}\,[\![x_9]\!]\,p_0\,\kappa_4\,s_4 =}$

$\mathsf{force}\, x_9\, \kappa_4\, s_4 =$

$\mathcal{E}\,[\![4]\!]\,p_0\,\kappa_5\,s_5 =$

$\kappa_5 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_4\,\varepsilon\,\langle\rho \oplus \{x_9 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_5 = \langle\rho_4 \oplus \{x_9 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_4\rangle$

$\kappa_5\,\langle 4, \emptyset\rangle\,s_5 =$

$\kappa_4\,\langle 4, \emptyset\rangle\,\langle\rho_5 \oplus \{x_9 \mapsto \langle 4, \emptyset\rangle\}, \mathrm{sch}_5\rangle =$

$\kappa_3\,\langle 4, \emptyset\rangle\,\langle\rho_4 \oplus \{x_9 \mapsto \langle 4, \emptyset\rangle, x_7 \mapsto \langle 4, \emptyset\rangle\}, \mathrm{sch}_4\rangle =$

$\kappa_2\,\langle 4, \emptyset\rangle\,\langle\rho_3 \oplus \{x_9 \mapsto \langle 4, \emptyset\rangle, x_7 \mapsto \langle 4, \emptyset\rangle, x_6 \mapsto \langle 4, \emptyset\rangle\}, \mathrm{sch}_3\rangle =$

$\kappa_1\,\langle 4, \emptyset\rangle\,\langle\rho_2 \oplus \{x_9 \mapsto \langle 4, \emptyset\rangle, x_7 \mapsto \langle 4, \emptyset\rangle, x_6 \mapsto \langle 4, \emptyset\rangle\}, \mathrm{sch}_2\rangle =$

$\kappa_0\,\langle 4, \emptyset\rangle\,\langle\rho_1 \oplus \{x_9 \mapsto \langle 4, \emptyset\rangle, x_7 \mapsto \langle 4, \emptyset\rangle, x_6 \mapsto \langle 4, \emptyset\rangle,$
$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle, x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle, main \mapsto \langle 4, \emptyset\rangle\}, \mathrm{sch}_1\rangle =$

$\boxed{\{\langle\rho_0 \oplus \{x_6 \mapsto \langle 4, \emptyset\rangle, x_7 \mapsto \langle 4, \emptyset\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle,$
$x_9 \mapsto \langle 4, \emptyset\rangle, x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle, main \mapsto \langle 4, \emptyset\rangle\}, \emptyset\rangle\}}$

(1.2) $\boxed{\mathcal{E}\,[\![x_{10}]\!]\,p_0\,\kappa_4\,s_4 =}$

$\mathsf{force}\, x_{10}\, \kappa_4\, s_4 =$

$\mathcal{E}\,[\![4]\!]\,p_0\,\kappa_6\,s_6 =$

$$\kappa_6 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_4\ \varepsilon\ \langle\rho \oplus \{x_{10} \mapsto \varepsilon\}, \mathrm{sch}\rangle$$
$$s_6 = \langle\rho_4 \oplus \{x_{10} \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_4\rangle$$

$$\kappa_5\ \langle 5, \emptyset\rangle\ s_6 =$$

$$\kappa_4\ \langle 5, \emptyset\rangle\ \langle\rho_5 \oplus \{x_{10} \mapsto \langle 4, \emptyset\rangle\}, \mathrm{sch}_6\rangle =$$

$$\kappa_3\ \langle 5, \emptyset\rangle\ \langle\rho_4 \oplus \{x_{10} \mapsto \langle 5, \emptyset\rangle, x_7 \mapsto \langle 5, \emptyset\rangle\}, \mathrm{sch}_4\rangle =$$

$$\kappa_2\ \langle 5, \emptyset\rangle\ \langle\rho_3 \oplus \{x_{10} \mapsto \langle 5, \emptyset\rangle, x_7 \mapsto \langle 5, \emptyset\rangle, x_6 \mapsto \langle 5, \emptyset\rangle\}, \mathrm{sch}_3\rangle =$$

$$\kappa_1\ \langle 5, \emptyset\rangle\ \langle\rho_2 \oplus \{x_{10} \mapsto \langle 5, \emptyset\rangle, x_7 \mapsto \langle 5, \emptyset\rangle, x_6 \mapsto \langle 5, \emptyset\rangle\}, \mathrm{sch}_2\rangle =$$

$$\kappa_0\ \langle 5, \emptyset\rangle\ \langle\rho_1 \oplus \{x_{10} \mapsto \langle 5, \emptyset\rangle, x_7 \mapsto \langle 4, \emptyset\rangle, x_6 \mapsto \langle 5, \emptyset\rangle,$$
$$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle, x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle, \mathit{main} \mapsto \langle 5, \emptyset\rangle\}, \mathrm{sch}_1\rangle =$$

$$\boxed{\{\langle\rho_0 \oplus \{x_6 \mapsto \langle 5, \emptyset\rangle, x_7 \mapsto \langle 5, \emptyset\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle,\ x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle, x_{10} \mapsto \langle 5, \emptyset\rangle, \mathit{main} \mapsto \langle 5, \emptyset\rangle\}, \emptyset\rangle\}}$$

(2) $\boxed{\mathcal{E}\,[\![x_8]\!]\ p_0\ \kappa_3\ s_3 =}$

$$\mathsf{force}\ x_8\ \kappa_3\ s_3 =$$

$$\mathcal{E}\,[\![3]\!]\ p_0\ \kappa_7\ s_7 =$$
$$\kappa_7 = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sch}\rangle.\kappa_3\ \varepsilon\ \langle\rho \oplus \{x_8 \mapsto \varepsilon\}, \mathrm{sch}\rangle$$
$$s_7 = \langle\rho_3 \oplus \{x_8 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_3\rangle$$

$$\kappa_7\ \langle 3, \emptyset\rangle\ s_7 =$$

$$\kappa_3\ \langle 3, \emptyset\rangle\ \langle\rho_7 \oplus \{x_8 \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sch}_7\rangle =$$

$$\kappa_2\ \langle 3, \emptyset\rangle\ \langle\rho_3 \oplus \{x_8 \mapsto \langle 3, \emptyset\rangle, x_6 \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sch}_3\rangle =$$

$$\kappa_1\ \langle 3, \emptyset\rangle\ \langle\rho_2 \oplus \{x_8 \mapsto \langle 3, \emptyset\rangle, x_6 \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sch}_2\rangle =$$

$$\kappa_0\ \langle 3, \emptyset\rangle\ \langle\rho_1 \oplus \{x_8 \mapsto \langle 3, \emptyset\rangle, x_6 \mapsto \langle 3, \emptyset\rangle,$$
$$x_7 \mapsto \langle p_0, \mathcal{E}\,[\![x_9 \bowtie x_{10}]\!]\,\rangle, x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle, x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle, \mathit{main} \mapsto \langle 3, \emptyset\rangle\}, \mathrm{sch}_1\rangle =$$

$$\boxed{\{\langle\rho_0 \oplus \{x_6 \mapsto \langle 3, \emptyset\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![x_9 \bowtie x_{10}]\!]\,\rangle, x_8 \mapsto \langle 3, \emptyset\rangle,\ x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle, x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle, \mathit{main} \mapsto \langle 3, \emptyset\rangle\}, \emptyset\rangle\}}$$

Por lo que el valor denotacional obtenido para la expresión de partida está formado por tres estados en los que el conjunto de canales es vacío:

Entorno$_1$

$x_6 \mapsto \langle 4, \emptyset\rangle$
$x_7 \mapsto \langle 4, \emptyset\rangle$
$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle$
$x_9 \mapsto \langle 4, \emptyset\rangle$
$x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle$
$\mathit{main} \mapsto \langle 4, \emptyset\rangle$

Entorno$_2$

$x_6 \mapsto \langle 5, \emptyset\rangle$
$x_7 \mapsto \langle 5, \emptyset\rangle$
$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle$
$x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle$
$x_{10} \mapsto \langle 5, \emptyset\rangle$
$\mathit{main} \mapsto \langle 5, \emptyset\rangle$

Entorno$_3$

$x_6 \mapsto \langle 3, \emptyset\rangle$
$x_7 \mapsto \langle p_0, \mathcal{E}\,[\![x_9 \bowtie x_{10}]\!]\,\rangle$
$x_8 \mapsto \langle 3, \emptyset\rangle$
$x_9 \mapsto \langle p_0, \mathcal{E}\,[\![4]\!]\,\rangle$
$x_{10} \mapsto \langle p_0, \mathcal{E}\,[\![5]\!]\,\rangle$
$\mathit{main} \mapsto \langle 3, \emptyset\rangle$

En este ejemplo hemos podido observar cómo se generan los conjuntos de sistemas cuando se evalúa una expresión no-determinista. La evaluación se estructura como el árbol de la Figura 8.4.

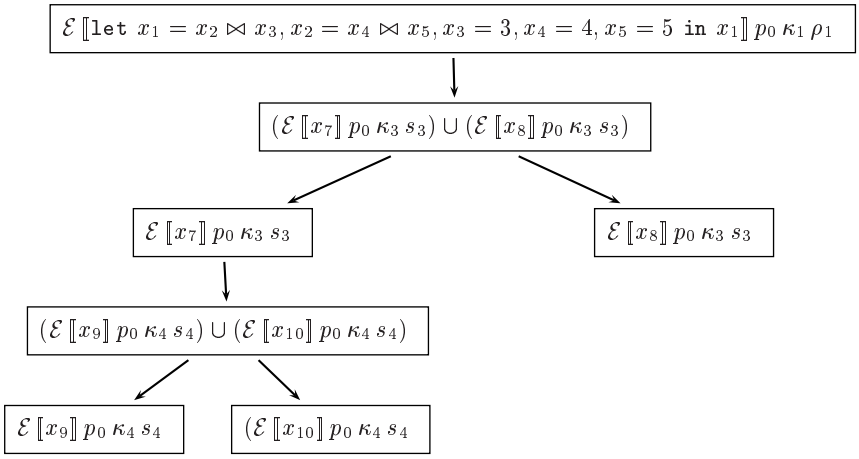


Figura 8.4: Ejemplo de ramificación en caso de no-determinismo

En este ejemplo se rompe la linealidad en la obtención del valor denotacional. La primera ruptura es provocada por la evaluación de $x_7 \bowtie x_8$; la segunda se produce en la rama de evaluación de x_7, al procederse con la evaluación de $x_9 \bowtie x_{10}$. Esta doble división da lugar a que el estado inicial haya sido transformado en un conjunto con tres estados finales. Se observa que en cada uno de estos sistemas el valor final de la variable *main* es distinto: $\langle 4, \emptyset \rangle$, $\langle 5, \emptyset \rangle$ y $\langle 3, \emptyset \rangle$

□

En la sección que sigue estudiamos la verificación de algunas propiedades de Jauja.

8.1.5. Propiedades

La semántica que acabamos de definir puede ser empleada para demostrar propiedades de las expresiones del lenguaje. En particular, los teoremas siguientes tratan la idempotencia, la conmutatividad y la asociatividad del operador no determinista $\bowtie$. Las primeras se verifican sin ninguna restricción, pero la última solamente se verifica en relación al valor de expresión final.

Antes de enunciar y demostrar los teoremas, introduciremos una condición necesaria para que no se produzcan interferencias por causa de la incorporación de variables nuevas en el proceso de normalización.

Definición 8.2 Decimos que x' es una *variable fresca con respecto a otra* x si, dados una continuación de expresión κ y un estado $s = \langle \rho, \mathrm{sch} \rangle$, $\mathcal{E}\,[\![x]\!]\,\kappa\,(s \oplus \{x' \mapsto \mathrm{not\_ready}\})$ devuelve un estado $s' = \langle \rho', \mathrm{sch}' \rangle$ tal que

1. $\rho'\,x = \varepsilon_x$, o bien
2. $\rho'\,x = \mathrm{undefined}$, o bien
3. $\rho'\,x = \mathrm{not\_ready}$, no siendo provocada esta asociación por el hecho de no estar lista x'.

Obsérvese que de estas condiciones se deduce que cuando $\rho\,x_i = \langle p, \nu \rangle$, x_i no va a ser forzada, y que si $\rho\,x_i = \varepsilon_i$, la variable no va a ser usada, en la evaluación de x.

□

Cuando escribamos, por ejemplo, que x_1, x_2, x_3 son frescas con respecto a y_1, y_2 nos referiremos a que cada una de las primeras por separado lo es con respecto a cada una de las segundas también por separado.

Proposición 8.1 *Para cualesquiera p_0 proceso, κ_0 continuación de expresión y s_0 estado, se verifica*

$$\mathcal{E}\,[\![x \bowtie x]\!]\,p_0\,\kappa_0\,s_0 = \mathcal{E}\,[\![x]\!]\,p_0\,\kappa_0\,s_0.$$

Demostración P.8.1
Calcularemos los estados finales en ambos casos:

$$\mathcal{E}\,[\![x \bowtie x]\!]\,p_0\,\kappa_0\,s_0 = (\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_0\,s_0) \cup (\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_0\,s_0) = \mathcal{E}\,[\![x]\!]\,p_0\,\kappa_0\,s_0.$$

El último paso es posible por la idempotencia de la unión conjuntista.

c.q.d.

Procedamos ahora con la conmutatividad.

Proposición 8.2 *Para cualesquiera p_0 proceso, κ_0 continuación de expresión y s_0 estado, se verifica*

$$\mathcal{E}\,[\![x_1 \bowtie x_2]\!]\,p_0\,\kappa_0\,s_0 = \mathcal{E}\,[\![x_2 \bowtie x_1]\!]\,p_0\,\kappa_0\,s_0$$

Demostración P.8.2
Calculemos los estados finales en ambos casos:

$\mathcal{E}\,[\![x_1 \bowtie x_2]\!]\,p_0\,\kappa_0\,s_0 =$	$\mathcal{E}\,[\![x_2 \bowtie x_1]\!]\,p_0\,\kappa_0\,s_0 =$
$(\mathcal{E}\,[\![x_1]\!]\,p_0\,\kappa_0\,s_0) \cup (\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_0\,s_0)$	$(\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_0\,s_0) \cup (\mathcal{E}\,[\![x_1]\!]\,p_0\,\kappa_0\,s_0)$

La conmutatividad de la unión de conjuntos garantiza la igualdad.

c.q.d.

Con el siguiente teorema demostraremos que el operador $\bowtie$ es asociativo, desde el punto de vista del valor final. Evidentemente no es cierto que lo sea en relación a los estados alcanzados, pues la normalización obliga a introducir variables intermedias que tomarán valores distintos dependiendo de cuál sea la rama de $\bowtie$ elegida.

Teorema 8.1 *Sean p_0 un proceso y s_0 un estado. Sean x_0, x_1, x_5 y x_6 variables frescas con respecto a x_2, x_3 y x_4, y sea κ_0 la continuación de expresión $\lambda\varepsilon.\lambda s.\{\langle\{valor \mapsto \varepsilon\}, \emptyset\rangle\}$. Entonces se tiene:*

$$\mathcal{E}\,[\![\texttt{let } x_0 = x_3 \bowtie x_4, x_1 = x_2 \bowtie x_0 \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0 = \mathcal{E}\,[\![\texttt{let } x_0 = x_2 \bowtie x_3, x_1 = x_0 \bowtie x_4 \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0$$

Demostración T.8.1
Para demostrar el resultado calculemos ambos valores:

$\mathcal{E}\,[\![\texttt{let } x_0 = x_3 \bowtie x_4, x_1 = x_2 \bowtie x_0 \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0 =$
$\mathcal{E}\,[\![x_6]\!]\,p_0\,\kappa_1\,s_1 =$
$\kappa_1 = \kappa_0$
$s_1 = \langle \rho_0 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![x_3 \bowtie x_4]\!]\,, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![x_2 \bowtie x_5]\!]\,\rangle\}, \mathrm{sch}_0\rangle$
$\mathsf{force}\,x_6\,\kappa_1\,s_1 =$
$\mathcal{E}\,[\![x_2 \bowtie x_5]\!]\,p_0\,\kappa_2\,s_2 =$
$\kappa_2 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle \rho \oplus \{x_6 \mapsto \varepsilon\}, \mathrm{sch}\rangle =$
$\lambda\varepsilon.\lambda s.\{\langle\{valor \mapsto \varepsilon\}, \emptyset\rangle\} = \kappa_0$
$s_2 = \langle \rho_1 \oplus \{x_6 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_1\rangle =$
$\boxed{(\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_0\,s_2)} \cup (\mathcal{E}\,[\![x_5]\!]\,p_0\,\kappa_0\,s_2)$

Calculemos el valor de la segunda variable:

$\mathsf{force}\,x_5\,\kappa_0\,s_2 =$
$\mathcal{E}\,[\![x_3 \bowtie x_4]\!]\,p_0\,\kappa_3\,s_2 =$
$\kappa_3 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle \rho \oplus \{x_5 \mapsto \varepsilon\}, \mathrm{sch}\rangle$
$\lambda\varepsilon.\lambda s.\{\langle\{valor\}, \emptyset\rangle\} = \kappa_0$
$s_3 = \langle \rho_2 \oplus \{x_5 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$
$\boxed{(\mathcal{E}\,[\![x_3]\!]\,p_0\,\kappa_0\,s_3)} \cup \boxed{(\mathcal{E}\,[\![x_4]\!]\,p_0\,\kappa_0\,s_3)}$

$\mathcal{E}\,[\![\texttt{let } x_0 = x_2 \bowtie x_3, x_1 = x_0 \bowtie x_4 \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0 =$
$\mathcal{E}\,[\![x_6]\!]\,p_0\,\kappa_1\,s_1 =$
$\kappa_1 = \kappa_0$
$s_1 = \langle \rho_0 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![x_2 \bowtie x_3]\!]\,, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![x_5 \bowtie x_4]\!]\,\rangle\}, \mathrm{sch}_0\rangle$
$\mathsf{force}\,x_6\,\kappa_1\,s_1 =$
$\mathcal{E}\,[\![x_5 \bowtie x_4]\!]\,p_0\,\kappa_2\,s_2 =$
$\kappa_2 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle \rho \oplus \{x_6 \mapsto \varepsilon\}, \mathrm{sch}\rangle =$
$\lambda\varepsilon.\lambda s.\{\langle\{valor \mapsto \varepsilon\}, \emptyset\rangle\} = \kappa_0$
$s_2 = \langle \rho_1 \oplus \{x_6 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_1\rangle =$
$(\mathcal{E}\,[\![x_5]\!]\,p_0\,\kappa_0\,s_2) \cup \boxed{(\mathcal{E}\,[\![x_4]\!]\,p_0\,\kappa_0\,s_2)}$

Calculemos el valor de la primera variable:

$\mathsf{force}\,x_5\,\kappa_0\,s_2 =$
$\mathcal{E}\,[\![x_2 \bowtie x_3]\!]\,p_0\,\kappa_3\,s_2 =$
$\kappa_3 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle \rho \oplus \{x_5 \mapsto \varepsilon\}, \mathrm{sch}\rangle$
$\lambda\varepsilon.\{\lambda s.\langle\{valor\}, \emptyset\rangle\} = \kappa_0$
$s_3 = \langle \rho_2 \oplus \{x_5 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$
$\boxed{(\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_0\,s_3)} \cup \boxed{(\mathcal{E}\,[\![x_3]\!]\,p_0\,\kappa_0\,s_3)}$

Analicemos el caso de una variable, x_i $(2 \leq i \leq 4)$. Centrémonos en el caso de que la evaluación de dicha variable no es errónea, es decir, produce un valor de expresión ε_i (este valor es independiente de si la evaluación es sobre s_2 o s_3, pues estos estados difieren solamente en las variables frescas con respecto a x_i $(2 \leq i \leq 4)$). Entonces se procederá a aplicar la continuación de expresión inicial a dicho valor y al sistema que hubiera resultado de la evaluación, devolviendo $\{\langle\{valor \mapsto \varepsilon_i\}, \emptyset\rangle\}$. Teniendo todo esto en consideración, se tienen las igualdades siguientes:

$$(\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_0\,s_2) \cup ((\mathcal{E}\,[\![x_3]\!]\,p_0\,\kappa_0\,s_3) \cup (\mathcal{E}\,[\![x_4]\!]\,p_0\,\kappa_0\,s_3)) =$$
$$\{\langle\{valor \mapsto \varepsilon_2\}, \emptyset\rangle\} \cup (\{\langle\{valor \mapsto \varepsilon_3\}, \emptyset\rangle\} \cup \{\langle\{valor \mapsto \varepsilon_4\}, \emptyset\rangle\})$$

$$((\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_0\,s_3) \cup (\mathcal{E}\,[\![x_3]\!]\,p_0\,\kappa_0\,s_3)) \cup (\mathcal{E}\,[\![x_4]\!]\,p_0\,\kappa_0\,s_2)) =$$
$$(\{\langle\{valor \mapsto \varepsilon_2\}, \emptyset\rangle\} \cup \{\langle\{valor \mapsto \varepsilon_3\}, \emptyset\rangle\}) \cup \{\langle\{valor \mapsto \varepsilon_4\}, \emptyset\rangle\}$$

y la asociatividad de la unión de conjuntos asegura el resultado. c.q.d.

Empleando el Teorema 8.1 se puede demostrar la idempotencia de $\bowtie$ con respecto al primer argumento:

Teorema 8.2 *Sean p_0 un proceso y s_0 un estado. Sean x_0, x_1, x_2 y x_3 variables frescas con respecto a x e y, y sea κ_0 la continuación de expresión $\lambda\varepsilon.\lambda s.\{\langle\{valor \mapsto \varepsilon\}, \emptyset\rangle\}$. Entonces se tiene:*

$$\mathcal{E}\,[\![\texttt{let } x_0 = x \bowtie y, x_1 = x \bowtie x_0 \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0 = \mathcal{E}\,[\![x \bowtie y]\!]\,p_0\,\kappa_0\,s_0.$$

Demostración T.8.2
La demostración del resultado pasa por calcular ambos valores:

$\mathcal{E}\,[\![\texttt{let } x_0 = x \bowtie y, x_1 = x \bowtie x_0 \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0 =$ (por Teorema 8.1)
$\mathcal{E}\,[\![\texttt{let } x_0 = x \bowtie x, x_1 = x_0 \bowtie y \texttt{ in } x_1]\!]\,p_0\,\kappa_0\,s_0 =$
$\mathcal{E}\,[\![x_3]\!]\,p_0\,\kappa_1\,s_1 =$
  $\kappa_1 = \kappa_0$
  $s_1 = \langle\rho_0 \oplus \{x_2 \mapsto \langle p_0, \mathcal{E}\,[\![x \bowtie x]\!]\,\rangle\}, \mathrm{sdh}_0\rangle$
$\mathsf{force}\,x_3\,\kappa_1\,s_1 =$
$\mathcal{E}\,[\![x_2 \bowtie y]\!]\,p_0\,\kappa_2\,s_2 =$
  $\kappa_2 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle\rho \oplus \{x_3 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$
  $s_2 = \langle\rho_1 \oplus \{x_3 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_1\rangle =$
$(\mathcal{E}\,[\![x_2]\!]\,p_0\,\kappa_2\,s_2) \cup (\mathcal{E}\,[\![y]\!]\,p_0\,\kappa_2\,s_2) =$
    Calculemos el valor de la primera variable:
    $\mathsf{force}\,x_2\,\kappa_2\,s_2 =$
    $\mathcal{E}\,[\![x \bowtie x]\!]\,p_0\,\kappa_3\,s_3 =$
      $\kappa_3 = \lambda\varepsilon.\lambda s.\kappa_2\,\varepsilon\,\langle\rho \oplus \{x_2 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$
      $s_3 = \langle\rho_2 \oplus \{x_2 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_2\rangle$
    $(\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_3\,s_3) \cup (\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_3\,s_3) =$
    $(\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_3\,s_3)$
$(\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_3\,s_3) \cup (\mathcal{E}\,[\![y]\!]\,p_0\,\kappa_2\,s_2)$
$(\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_0\,(\langle\rho_0 \oplus \{x_2 \mapsto \mathrm{not\_ready}, x_3 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_0\rangle)) \cup (\mathcal{E}\,[\![y]\!]\,p_0\,\kappa_0\,(\langle\rho_0 \oplus \{x_3 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_0\rangle))$

Por otra parte, el valor de la otra expresión es:

$\mathcal{E}\,[\![x \bowtie y]\!]\,p_0\,\kappa_0\,s_0 = (\mathcal{E}\,[\![x]\!]\,p_0\,\kappa_0\,s_0) \cup (\mathcal{E}\,[\![y]\!]\,p_0\,\kappa_0\,s_0)$

Como las variables x_2 y x_3 eran frescas con respecto a x e y, ambos conjuntos de estados son en realidad el mismo.

c.q.d.

Valgan los ejemplos anteriores como ilustración del tipo de propiedades que se suelen demostrar a partir de la definición de una semántica denotacional como la presente.

De forma similar se podría intentar demostrar la corrección de los axiomas y reglas de una potencial semántica algebraica de Jauja, con respecto a la presente semántica denotacional.

Pasamos, en la siguiente sección, a definir la semántica de Jauja cuando la comunicación vía canales se extiende con la posibilidad de emplear *streams*.

8.2. Jauja con comunicación vía *streams*

Recordemos la sintaxis (restringida) correspondiente a las estructuras sintácticas relacionadas con la existencia de *streams*:

E	$::=$	$\Lambda[x_1 : x_2].E_1 [\!] E_2$	Λ-abstracción
	$\mid$	L	lista
L	$::=$	`nil`	lista vacía
	$\mid$	$[x_1 : x_2]$	lista no vacía

El camino a seguir para definir la semántica denotacional de este subconjunto del lenguaje tendrá las etapas siguientes: adaptación de los dominios semánticos de la Figura 8.1, modificación de la función de evaluación incluida en la Figura 8.2, modificación de las funciones auxiliares de la Figura 8.3 e inclusión de nuevas funciones auxiliares.

8.2.1. Dominios semánticos

Todos los dominios empleados para este subconjunto de Jauja aparecen definidos en la Figura 8.5. Nos detendremos en la explicación de aquéllos que han sufrido modificaciones con respecto a los de la Figura 8.1 y de los que son introducidos para dar significado a las nuevas construcciones sintácticas.

Recordemos que hemos introducido listas con el objetivo de poder tener canales de comunicación con el comportamiento de *streams*. Además, un canal por el que se comunica un valor único es inutilizado o cerrado cuando se ha comunicado dicho valor por él; por el contrario, un canal del tipo *stream* está abierto hasta que la comunicación del valor nil provoca su cierre. La modelización denotacional de este comportamiento necesita de mecanismos que reflejen el estado abierto o cerrado del canal:

Abierto: la posibilidad de continuar comunicando valores por el canal se modeliza introduciendo en el valor denotacional del canal un identificador, o variable, que se corresponde en el entorno con la lista-*stream* obtenida previamente, o con la clausura que dará lugar a la lista-*stream*. Evidentemente, esta variable también podría quedar ligada a un valor de abstracción; en ese caso el canal quedaría cerrado tras esta comunicación.

Cerrado: Este estado del canal se modeliza mediante la sustitución del identificador por el valor especial closed.

	Cont	=	**State** → **SState**	continuaciones
$S \in$	**SState**	=	$\mathcal{P}_f$(**State**)	conjuntos de estados
$s \in$	**State**	=	**Env** × **SChan**	estados
$\kappa \in$	**ECont**	=	**EVal** → **Cont**	continuaciones de expresión
$\rho \in$	**Env**	=	**Ide** → (**Val** + {undefined})	entornos
$v \in$	**Val**	=	**EVal** + (**IdProc** × **Clo**) + {not_ready}	valores
$\varepsilon \in$	**EVal**	=	(**Abs** × **Ides**) + (**Ide** × **Ide**) + {nil}	valores de expresión
$\alpha \in$	**Abs**	=	**Ide** → **Clo**	valores de abstracción
$\nu \in$	**Clo**	=	**IdProc** → **ECont** → **Cont**	clausuras
sdh ∈	**SChan**	=	$\mathcal{P}_f$(**Chan**)	conjuntos de canales
	Chan	=	**IdProc** × (**Ide** + {closed}) × (**SCVal** + {<>}) × **IdProc**	canales
cv ∈	**CVal**	=	**Abs** + **CList**	valores comunicables
$\sigma \in$	**CList**	=	{nil} + (**CVal** × **CList**)	listas comunicables
scv ∈	**SCVal**	=	**CVal***	secuencias
$I \in$	**Ides**	=	$\mathcal{P}_f$(**Ide**)	conjuntos de identificadores
$p, q \in$	**IdProc**			identificadores de proceso
scω ∈	**SCVal** + {<>}			

Figura 8.5: Jauja básico con *streams*: dominios semánticos

Con estas consideraciones en mente, la representación de un canal de comunicación —la terna ⟨productor, valor, consumidor⟩— se extiende con la información indicada. Por otra parte, como se comunican *streams* potencialmente infinitos, el valor de un canal corresponde a una secuencia de valores construida a partir de valores comunicables (en la Figura 8.5) siendo inicialmente la secuencia vacía, <>.

La introducción de las listas motiva también cambios en los valores de expresión. Tras la normalización de la Sección 4.3, obtenemos expresiones en las que todas sus subexpresiones son compartidas; logro que se desea mantener en el presente marco denotacional, en el que pretendemos definir un lenguaje perezoso, amén de otros objetivos. Por ello, la modificación del dominio **EVal** viene motivada por la necesidad de incluir el valor correspondiente a la lista vacía, nil, y por la intención de recoger las listas perezosas no vacías. Estas últimas se representan mediante un par de identificadores, **Ide** × **Ide**, el primero de los cuales se liga a la clausura o valor de la cabeza, y el segundo se asocia en el entorno o a la clausura o al valor de la cola.

Los valores de comunicación **CVal** también cambian: en el caso de que el canal sea de valor único, el valor solamente podrá ser una abstracción, si bien, en el canal de comunicación se representará como un *stream* con un único elemento;[9] si, por contra, el canal es de comportamiento *stream*, entonces el valor del canal será toda una secuencia de valores. Estos valores están incluidos en el nuevo dominio **CList** de listas comunicables: cada una de estas listas puede ser o la lista vacía, nil, o una lista cuya cabeza es un valor

[9] Para comunicar solamente una lista, ésta tiene que ser la cabeza de un *stream* con un único elemento

de comunicación.

Una vez definido el dominio de valores de comunicación, podemos definir las secuencias de valores de este dominio que aparecen en los canales de comunicación. Para representar la existencia de *streams* infinitos, a cada canal se le asocia una de las secuencias de valores comunicables mencionadas, que serán los valores de comunicación enviados a través de un canal.

Una vez definidos los dominios semánticos, el siguiente paso es definir la función de evaluación.

8.2.2. Función de evaluación

Siguiendo la misma línea que en el capítulo dedicado a la semántica operacional, nos limitaremos a explicar los cambios con respecto a la semántica definida para Jauja básico en la Sección 8.1. De todos modos, en la Figura 8.6 se incluye la función de evaluación completa.

Nos detenemos primero en la *creación de procesos*, pues la incorporación de *streams* de comunicación y listas hace que tengan que realizarse algunos cambios en su definición. El motivo es que ahora un canal no es de "usar y tirar", sino que un canal-*stream* permanece abierto hasta que se han comunicado todos los valores. Cuando se crea un canal se señala la variable cuya evaluación dará lugar al (primer) valor a comunicar por el mismo. En el canal de entrada se tiene la variable i —que ya introdujimos en la definición dada en la Figura 8.2—, mientras que en el de salida se introduce la variable o a tal efecto.

Los "pasos" que se siguen en la evaluación de la creación de procesos son:

1. Forzar la evaluación de la variable correspondiente a la abstracción, x_1, así como las de todas sus dependencias libres.

2. La continuación de expresión κ' crea dos nuevos canales: uno suspenso en i y el otro en o, ambos con el valor inicial $<>$.

3. Evaluar la aplicación, la cual se encuentra ligada a la variable o. Como resultado obtenemos un valor que se comunica entonces desde el nuevo proceso al padre. Todo esto se hace por medio de una llamada a la función forceFV .

4. Tras evaluar la aplicación, si estamos en presencia de canales tipo *stream*, éstos se deben evaluar al completo. De nuevo se invoca a la función forceFV, que con ayuda de la continuación de expresión kstr x, fuerza reiteradamente cada componente del *stream* y las correspondientes dependencias libres.

Recordemos que el enfoque de *semántica máxima* provoca la evaluación completa de todos los canales. Por ello, la continuación de expresión fuerza la evaluación tanto de i

$$\mathcal{E}\,[\![x]\!]\,p\,\kappa = \mathsf{force}\;x\;\kappa$$

$$\mathcal{E}\,[\![\backslash x.E]\!]\,p\,\kappa = \kappa\langle\lambda x.\mathcal{E}\,[\![E]\!]\,,\mathsf{fv}(\backslash x.E)\rangle$$

$$\begin{array}{l}\mathcal{E}\,[\![x_1\;x_2]\!]\,p\,\kappa = \mathcal{E}\,[\![x_1]\!]\,p\,\kappa' \\ \quad \texttt{donde}\;\kappa' = \lambda\varepsilon.\lambda s.\texttt{case}\;\varepsilon\;\texttt{of} \\ \qquad\qquad \langle\alpha, I\rangle \in \mathbf{Abs}\times\mathbf{Ides} \longrightarrow (\alpha\;x_2)\,p\,\kappa\,s \\ \qquad\qquad \mathsf{e.o.c.} \longrightarrow \mathsf{wrong}\;s \\ \qquad\quad \texttt{endcase}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,[\![x_1\texttt{\#}x_2]\!]\,p\,\kappa = \mathsf{forceFV}\;x_1\;\kappa' \\ \quad \texttt{donde}\;\kappa' = \lambda\varepsilon.\lambda s.\texttt{case}\;\varepsilon\;\texttt{of} \\ \qquad\quad \langle\alpha, I\rangle \in \mathbf{Abs}\times\mathbf{Ides} \longrightarrow \mathsf{forceFV}\;o\;\kappa''\;s' \\ \qquad\qquad \texttt{donde}\;q = \mathsf{newIdProc}\;s \\ \qquad\qquad\qquad\quad o = \mathsf{newIde}\;s \\ \qquad\qquad\qquad\quad s' = s \oplus \langle\{o \mapsto \langle q, (\alpha\;i)\rangle, i \mapsto \langle p, \mathcal{E}\,[\![x_2]\!]\,\rangle\}, \{\langle p, i, <>, q\rangle, \langle q, o, <>, p\rangle\}\rangle \\ \qquad\qquad\qquad\quad \boxed{\begin{array}{l}\kappa''_{min} = \lambda\varepsilon'.\lambda s''.\kappa\,\varepsilon'\;s'' = \kappa \\ \hline \kappa''_{max} = \lambda\varepsilon'.\lambda s''.\bigcup_{s_o\in S_o}\kappa\,\varepsilon'\;s_o \\ \quad \texttt{donde}\;S_i = \mathsf{forceFV}\;i\,(\mathsf{kstr}\;i)\;s'' \\ \qquad\qquad S_o = \bigcup_{s_i\in S_i}\mathsf{forceFV}\;o\,(\mathsf{kstr}\;o)\;s_i\end{array}} \\ \qquad\quad \mathsf{e.o.c.} \longrightarrow \mathsf{wrong}\;s \\ \qquad \texttt{endcase}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,[\![\texttt{let}\;\{x_i = E_i\}_n\;\texttt{in}\;x]\!]\,p\,\kappa = \lambda\langle\rho, \mathrm{sdh}\rangle.\mathcal{E}\,[\![x]\!]\,p\,\kappa'\,\langle\rho', \mathrm{sdh}\rangle \\ \quad \texttt{donde}\;\{y_1, \ldots, y_n\} = \mathsf{newIde}\;n\;\rho \\ \qquad\quad \rho' = \rho \oplus \{y_i \mapsto \langle p, \mathcal{E}\,[\![E_i[y_1/x_1, \ldots, y_n/x_n]]\!]\,\rangle \mid 1 \le i \le n\} \\ \qquad\quad \boxed{\begin{array}{l}\kappa'_{min} = \lambda\varepsilon.\lambda\langle\rho, \mathrm{sdh}\rangle.\kappa\,\varepsilon\,s' \\ \quad \texttt{donde}\;I = \{y_i \mid E_i \equiv x_1^i\texttt{\#}x_2^i \wedge (\rho\;y_i) \in \mathbf{IdProc}\times\mathbf{Clo}\} \\ \qquad\quad m = \mathsf{card}\;I \\ \qquad\quad \{q_1, \ldots, q_m\} = \mathsf{newIdProc}\;m\;\langle\rho, \mathrm{sdh}\rangle \\ \qquad\quad s' = \langle\rho, \mathrm{sdh}\rangle \oplus_{ch} \{\langle p, <>, q_j\rangle, \langle q_j, <>, p\rangle \mid 1 \le j \le m\} \\ \hline \kappa'_{max} = \lambda\varepsilon.\lambda s.\bigcup_{s_f\in S_f}\kappa\,\varepsilon\,s_f \\ \quad \texttt{donde}\;I = \{y_i \mid E_i \equiv x_1^i\texttt{\#}x_2^i \wedge 1 \le i \le n\} \\ \qquad\quad S_f = \mathsf{mforce}\;I\;s\end{array}}\end{array}$$

$$\begin{array}{l}\mathcal{E}\,[\![\Lambda[y_1 : y_2].E_1\|E_2]\!]\,p\,\kappa = \\ \quad = \kappa\langle\lambda x.\lambda p'.\lambda\kappa'.\lambda s.\bigcup_{s_a\in S_a}\texttt{case}\;(\rho_a\;x)\;\texttt{of} \\ \qquad\quad \mathsf{nil} \longrightarrow \mathcal{E}\,[\![E_1]\!]\,p\,\kappa'\;s_a \\ \qquad\quad \langle z_1, z_2\rangle \longrightarrow \mathcal{E}\,[\![E_2[z_1/y_1, z_2/y_2]]\!]\,p\,\kappa'\;s_a \\ \qquad\quad \mathsf{e.o.c.} \longrightarrow \mathsf{wrong}\;s_a \\ \qquad \texttt{endcase}, \\ \qquad \mathsf{fv}(\Lambda[y_1 : y_2].E_1\|E_2)\rangle \\ \quad \texttt{donde}\;S_a = \mathsf{force}\;x\;id_\kappa\;s\end{array}$$

$$\mathcal{E}\,[\![\texttt{nil}]\!]\,p\,\kappa = \kappa\;\mathsf{nil}$$

$$\mathcal{E}\,[\![[x_1 : x_2]]\!]\,p\,\kappa = \kappa\,\langle x_1, x_2\rangle$$

$$\mathcal{E}\,[\![x_1 \bowtie x_2]\!]\,p\,\kappa = \lambda s.(\mathcal{E}\,[\![x_1]\!]\,p\,\kappa\,s) \cup (\mathcal{E}\,[\![x_2]\!]\,p\,\kappa\,s)$$

Figura 8.6: Jauja básico con *streams*: función de evaluación

como de o. ¿Qué sucede si un canal era un *stream* y parte de él ya se ha evaluado por demanda? De cara a los efectos sobre el estado final no cambia nada: el "re-forzado" de una variable no provoca cambios en el estado, simplemente se usa su valor ya disponible. Además, cada una de las componentes evaluadas que nos encontremos ya habrá sido comunicada, y la correspondiente variable no definiría ya canal alguno. En conclusión, este forzado de variables será efectivo cuando se llegue a la primera componente del *stream* que todavía no haya sido evaluada ni comunicada.

La continuación de expresión κ''_{min} difiere de la presentada en la Figura 8.2 en que en el presente caso no se entra en la disquisición de distinguir si el valor del canal de entrada ha sido necesario o no, puesto que esta tarea se realiza ahora vía forceFV y la información de la variable se obtiene de la definición del canal.

En lo que atañe a la *abstracción de listas*, la función de evaluación $\mathcal{E}$ se define de manera similar a como se hizo en el caso de la abstracción funcional. La única diferencia, debida a la estrictez de la segunda, es la manera de construir la abstracción: el argumento tiene que ser forzado (en caso de que se pase ya evaluado este forzado no tiene ningún efecto), y según su forma se determina la evaluación que procede: si el valor corresponde a una lista vacía, la clausura a devolver será la determinada por la expresión E_1; mientras que si el valor es una lista no vacía, la clausura se construye con E_2. Cualquier otro caso será interpretado como erróneo (wrong).

La evaluación de las *listas*, tanto *vacías* como *no vacías*, es ahora inmediata, pues solamente queda por aplicar al correspondiente valor denotacional la continuación de expresión que proceda.

8.2.3. Funciones semánticas auxiliares

Las nuevas funciones auxiliares necesarias para definir $\mathcal{E}$ quedan expuestas en las Figuras 8.7 y 8.8. Las primeras son para forzar las variables necesarias.[10] Las segundas sirven para tratar con *streams* y componer las secuencias de los canales de comunicación.

8.2.3.1. Funciones semánticas auxiliares para forzar

En primer lugar nos detenemos en explicar las funciones que fuerzan la evaluación de variables (Figura 8.7).

La primera diferencia que la función force presenta con respecto a la función definida en la Sección 8.1.3, es que ahora force no es ejecutiva en sí misma, sino que simplemente decide si hay que forzar la variable demandada, bien porque no sea de comunicación, en cuyo caso se llama a la función forceS, o porque se trate de una variable de comunicación, que aparece en la definición de algún canal del conjunto de canales del sistema. El forzado

[10] Las funciones mforce y mforceFV no aparecen aquí porque coinciden exactamente con su definición dada en la Figura 8.3.

```
force :: Ide → ECont → Cont
force x κ = λs.case ⟨p, x, cv q⟩ ∈ sdh of
  true ⟶ forceFV x κ
  false ⟶ forceS x κ
  endcase
```

```
forceS :: Ide → ECont → Cont
forceS x κ = λ⟨ρ, sdh⟩.case (ρ x) of
  ε ∈ EVal ⟶ κ ε ⟨ρ, sdh⟩
  ⟨p, ν⟩ ∈ (IdProc × Clo) ⟶ ν p κ′ s′
    donde κ′ = λε′.λ⟨ρ′, sdh′⟩.κ ε′ ⟨ρ′ ⊕ {x ↦ ε′}, sdh′⟩
           s′ = ⟨ρ ⊕ {x ↦ not_ready}, sdh⟩
  e.o.c. ⟶ wrong ⟨ρ, sdh⟩
  endcase
```

```
forceFV :: Ide → ECont → Cont
forceFV x κ = forceS x κ′
  donde κ′ = λε.λs.case ε of
    ⟨α, I⟩ ∈ Abs × Ides ⟶ ⋃_{s_f ∈ S_f} κ ε s′_f
        donde S_f = mforceFV I s
              s′_f = case ⟨p, x, scω, q⟩ ∈ sdh_f of
                false ⟶ s_f
                true ⟶ case ⟨p, x, scω, q⟩ of
                  ⟨p, x, <>, q⟩ ⟶ s ⊕ {⟨p, closed, α, q⟩}
                  ⟨p, x, scv, q⟩ ⟶ s_f ⊕_ch {⟨p, closed, scv ++ α, q⟩}
                endcase
              endcase
    nil ⟶ case ⟨p, x, scω, q⟩ ∈ sdh of
        false ⟶ κ nil s
        true ⟶ κ ε s′
          donde s′ = case ⟨p, x, scω, q⟩ ∈ sdh of
                ⟨p, x, <>, q⟩ ⟶ s ⊕ {⟨p, closed, nil, q⟩}
                ⟨p, x, scv, q⟩ ⟶ s ⊕ {⟨p, closed, scv ++ nil, q⟩}
              endcase
      endcase
    ⟨x_h, x_t⟩ ∈ Ide × Ide ⟶ case ⟨p, x, scω, q⟩ ∈ sdh of
        false ⟶ ⋃_{s_f ∈ S_f} κ ε s_f
          donde S_f = mforceFV {x_h, x_t} s
        true ⟶ ⋃_{s_h ∈ S_h} κ ⟨x_h, c⟩ s′_h
          donde S_h = forceFV x_h id_κ s
                s′_h = case cve_h of
                  cv_h ∈ CVal ⟶ case ⟨p, x, scω′, q⟩ ∈ sdh_h of
                      ⟨p, x, <>, q⟩ ⟶ s_h ⊕ ⟨{c ↦ ⟨p, ℰ ⟦x_t⟧ ⟩, x ↦ ⟨x_h, c⟩},
                                              {⟨p, c, cv_h, q⟩}⟩
                      ⟨p, x, scv, q⟩ ⟶ s_h ⊕ ⟨{c ↦ ⟨p, ℰ ⟦x_t⟧ ⟩, x ↦ ⟨x_h, c⟩},
                                              {⟨p, c, scv ++ cv_h, q⟩}⟩
                    endcase
                  e.o.c. ⟶ wrong s_h
                endcase
                c = newIde s_h
                cve_h = compval x_h s_h
      endcase
  endcase
```

Figura 8.7: Jauja básico con *streams*: funciones semánticas auxiliares para forzar

tiene que extenderse a las dependencias libres, para lo que se invoca a la función forceFV. Esta distinción de casos viene motivada porque la creación de procesos no es el único punto donde se demanda la evaluación de un canal, como sucedía en el caso de Jauja básico de la Sección 8.1, pues tras la creación de un *stream* éste puede quedar abierto y ser demandado para la evaluación de otra expresión cualquiera.

Una vez separadas estas funciones, reparamos en que forceS es idéntica a la función force de la Sección 8.1.3, como por otra parte cabía esperar.

Es en la función forceFV donde aparecen los principales detalles derivados de la evaluación de *streams*. En todos los casos hay que distinguir si la variable es o no de comunicación. Tras forzar una variable se pueden dar los tres casos derivados de la construcción de **EVal**:

1. El valor obtenido es una abstracción: como se demandaba la evaluación de sus variables libres, se procede a ello con la llamada recursiva adecuada. Seguidamente, si la variable era de comunicación, se comunica la abstracción y se cierra el canal. El punto final de este caso consiste en aplicar la continuación de expresión al valor obtenido en cada uno de los estados resultado de los pasos anteriores.

2. El valor obtenido es nil: si no hay que comunicarlo, simplemente se aplica la continuación de expresión a este valor y al estado pasado como argumento. Pero si la variable es de comunicación, hay que cerrar el canal.

3. El valor es una lista no vacía: en este caso las diferencias son más sustanciales. Si la variable no es de comunicación esta lista se debe evaluar hasta forma normal, por lo que se procede a forzar la evaluación tanto de la cabeza como de la cola, y, por supuesto, se extiende la demanda a todas las dependencias libres. Sin embargo, si la variable es de comunicación, estamos ante la presencia de un *stream*, en cuyo caso la manera de proceder es forzar la cabeza —con sus dependencias incluidas—, comunicar la misma, y sustituir el identificador de comunicación del canal por uno nuevo que se asocia a la cola del *stream*.

Una vez descritas las funciones que fuerzan la evaluación de variables quedan por definir las funciones auxiliares que se precisan debido a la presencia de *streams*.

8.2.3.2. Funciones semánticas auxiliares para *streams*

La función ++ se emplea en la función mforceFV y se encarga de incorporar un nuevo elemento a la secuencia de valores comunicados por el canal-*stream*, de manera que la nueva secuencia es la concatenación de la existente con el nuevo elemento comunicado. Esta secuencia lleva cuenta del orden temporal del envío de valores por dicho canal-*stream*.

La función compval (composición de valor) compone el valor comunicado, mientras que la función kstr (continuación de expresión de *streams*) continúa con la evaluación

de un *stream*. En la Figura 8.8 se muestra la definición de estas dos últimas funciones auxiliares.

```
compval :: Ide → State → CValE
compval x s = case (ρ x) of
  ⟨α, I⟩ ⟶ α
  nil ⟶ nil
  ⟨x1, x2⟩ ⟶ ⟨(compval x1 s), (compval x2 s)⟩
 endcase

kstr :: Ide → ECont
kstr x = λε.λs.case (ρ x) of
          ⟨α, I⟩ ∈ Abs × Ides ⟶ {s}
          nil ⟶ {s}
          ⟨xh, xt⟩ ∈ Ide × Ide ⟶ forceFV xh (kstr xt) s
         endcase
```

Figura 8.8: Jauja básico con *streams*: funciones semánticas auxiliares para *streams*

Como es posible que el valor comunicado a través de un *stream* sea una lista —evaluada a forma normal—, para añadir esta lista al conjunto de cadenas del canal hay que componerla, es decir, formar una lista de valores comunicables. Ésta es la misión de la función compval. En principio esta lista pertenece al dominio

$$\mathbf{CValE} = \mathbf{Abs} + \mathbf{CList} + \{\mathsf{nil}\} + (\mathbf{CValE} \times \mathbf{CValE})$$

que, en particular, contiene al dominio **CList**. Pero dicha lista puede ser "amorfa", es decir, no perteneciente a **CList**, al no terminar con la lista vacía. Al igual que se hizo en el caso de la semántica operacional (Capítulo 6), no permitiremos comunicar una lista que no esté bien formada.

Finalmente, en la definición de creación de procesos se utiliza una continuación de expresión especial, kstr, cuya misión es la siguiente:

- Si la variable no se encuentra ligada a una lista no vacía, entonces actúa como la continuación de expresión id_κ.
- Si la variable se halla ligada a una lista no vacía, se fuerza la evaluación de la cabeza y se extiende este forzado a la cola.

Concluye aquí la definición de la semántica denotacional para Jauja con no-determinismo simple y *streams*.

8.2.4. Ejemplos

Como en ocasiones anteriores, pasamos ahora a desarrollar algunos ejemplos que ilustran el cálculo del valor denotacional de algunas expresiones que consideramos representativas; en este caso las hemos escogido por mostrar comportamientos asociados al uso de listas y *streams*.

En primer lugar vamos a considerar una expresión en la que se demanda la evaluación de una lista, pero solamente hasta *whnf*.

Ejemplo 8.8 *Evaluación de una lista a whnf.*

En este caso la expresión de Jauja que vamos a considerar es:

$$\texttt{let } x_0 = [x_1 : x_2], x_1 = x_3\, x_3, x_2 = \texttt{nil}, x_3 = \backslash x.x, x_4 = x_3\, x_0 \texttt{ in } x_4.$$

Obsérvese que el cuerpo de la expresión `let` demanda la evaluación de la variable x_4, que a su vez es una aplicación. La lista $[x_1 : x_2]$ está ligada a la variable argumento de dicha aplicación. La abstracción es la identidad, por lo que la lista tendrá que evaluarse, pero no se evaluarán ni x_1 ni x_2, pues solamente evaluaremos hasta *whnf*.

El entorno inicial es:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus \{main \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{let } x_0 = [x_1 : x_2], x_1 = x_3\, x_3, x_2 = \texttt{nil}, x_3 = \backslash x.x, x_4 = x_3\, x_0 \texttt{ in } x_4]\!]\,\rangle\}$$

y el cálculo del valor denotacional a partir de *main* se desarrolla como sigue:

$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$

$\mathsf{force}\; main\,\kappa_0\,s_0 =$

$\mathcal{E}\,[\![\texttt{let } x_0 = [x_1 : x_2], x_1 = x_3\, x_3, x_2 = \texttt{nil}, x_3 = \backslash x.x, x_4 = x_3\, x_0 \texttt{ in } x_4]\!]\,p_0\,\kappa_1\,s_1 =$

$\kappa_1 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle \rho \oplus \{main \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_1 = \langle \rho_0 \oplus \{main \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_0\rangle$

$\mathcal{E}\,[\![x_9]\!]\,p_0\,\kappa_2\,s_2 =$

$\kappa_2 = \kappa_1$ [Porque no hay creaciones de proceso]

$s_2 = \langle \rho_1 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![[x_6 : x_7]]\!]\,\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![x_8\, x_8]\!]\,\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{nil}]\!]\,\rangle,$
$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![\backslash x.x]\!]\,\rangle, x_9 \mapsto \langle p_0, \mathcal{E}\,[\![x_8\, x_5]\!]\,\rangle\}, \mathrm{sdh}_1\rangle$

$\mathsf{force}\; x_9\,\kappa_2\,s_2 =$

$\mathcal{E}\,[\![x_8\, x_5]\!]\,p_0\,\kappa_3\,s_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda x.\kappa_2\,\varepsilon\,\langle \rho \oplus \{x_9 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_3 = \langle \rho_2 \oplus \{x_9 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_2\rangle$

$\mathcal{E}\,[\![x_8]\!]\,p_0\,\kappa_4\,s_4 =$

$\kappa_4 = \lambda\langle\alpha, I\rangle.\lambda s.(\alpha\, x_5)\,p_0\,\kappa_3\,s$

A la vista de la clausura ligada a x_8, simplificamos la continuación de expresión.

$s_4 = s_3$

$\mathsf{force}\; x_8\,\kappa_4\,s_4 =$

$\mathcal{E}\,[\![\backslash x.x]\!]\,p_0\,\kappa_5\,s_5 =$

$\kappa_5 = \lambda\varepsilon.\lambda x.\kappa_4\,\varepsilon\,\langle \rho \oplus \{x_8 \mapsto \varepsilon\}, \mathrm{sdh}\rangle$

$s_5 = \langle \rho_4 \oplus \{x_8 \mapsto \mathrm{not\_ready}\}, \mathrm{sdh}_4\rangle$

$\kappa_5\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_5 =$

$\kappa_4\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_6 =$

$$s_6 = \langle \rho_4 \oplus \{x_8 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch}_4\rangle$$

$\mathcal{E}\,[\![x_5]\!]\,p_0\,\kappa_3\,s_6 =$

$\mathsf{force}\,x_5\,\kappa_3\,s_6 =$

$\mathcal{E}\,[\![[x_6 : x_7]]\!]\,p_0\,\kappa_6\,s_7 =$

$$\kappa_6 = \lambda\varepsilon.\lambda x.\kappa_3\,\varepsilon\,\langle \rho \oplus \{x_5 \mapsto \varepsilon\}, \mathrm{sch}\rangle$$

$$s_7 = \langle \rho_6 \oplus \{x_5 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_6\rangle$$

$\kappa_6\,\langle x_6, x_7\rangle\,s_7 =$

$\kappa_3\,\langle x_6, x_7\rangle\,\langle \rho_6 \oplus \{x_5 \mapsto \langle x_6, x_7\rangle\}, \mathrm{sch}_6\rangle =$

$\kappa_2\,\langle x_6, x_7\rangle\,\langle \rho_6 \oplus \{x_5 \mapsto \langle x_6, x_7\rangle, x_9 \mapsto \langle x_6, x_7\rangle\}, \mathrm{sch}_6\rangle =$

$\kappa_0\,\langle x_6, x_7\rangle\,\langle \rho_6 \oplus \{x_5 \mapsto \langle x_6, x_7\rangle, x_9 \mapsto \langle x_6, x_7\rangle, main \mapsto \langle x_6, x_7\rangle\}, \mathrm{sch}_6\rangle =$

$\{\langle \{x_5 \mapsto \langle x_6, x_7\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![x_8\ x_8]\!]\,\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![\mathtt{nil}]\!]\,\rangle, x_8 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_9 \mapsto \langle x_6, x_7\rangle,$

$main \mapsto \langle x_6, x_7\rangle\}, \emptyset\rangle\}$

En el único estado final devuelto se observa que ni la cabeza ni la cola de la lista demandada han sido evaluados: tanto la una como la otra permanecen asociadas a clausuras, es decir, la lista ha sido evaluada a *whnf*.

□

En contraste con el Ejemplo 8.8, que acabamos de desarrollar, en el siguiente ejemplo la lista pasa a ser un *stream*, por tratarse del argumento de una creación de proceso.

Ejemplo 8.9 *Lista-stream.*

La expresión de Jauja que vamos a considerar es:

$$\mathtt{let}\ x_0 = [x_1 : x_2], x_1 = x_3\,x_3, x_2 = \mathtt{nil}, x_3 = \backslash x.x, x_4 = x_3\#x_0\ \mathtt{in}\ x_4$$

En este caso la evaluación demanda la creación de un proceso cuyo argumento es una lista, lo que se traduce en que dicha lista no se comportará como tal, sino que será un *stream* de comunicación.

El valor del entorno inicial es:

$$\rho_0 = \{x_i \mapsto \mathrm{undefined}\} \oplus$$
$$\{main \mapsto \langle p_0, \mathcal{E}\,[\![\mathtt{let}\ x_0 = [x_1 : x_2], x_1 = x_3\,x_3, x_2 = \mathtt{nil}, x_3 = \backslash x.x, x_4 = x_3\#x_0\ \mathtt{in}\ x_4]\!]\,\rangle\}$$

y a partir de *main*, el cálculo del valor denotacional se desarrolla como sigue:[11]

$\mathcal{E}\,[\![main]\!]\,p_0\,\kappa_0\,s_0 =$

[11] En algunos casos se omitirá la descripción explícita de alguna continuación de expresión, por no ser relevantes los cambios con respecto a la definición original y sí entorpecer, por su extensión, el seguimiento del ejemplo.

$\mathsf{force}\ main\,\kappa_0\ s_0 =$

$\mathcal{E}\,[\![\texttt{let}\ x_0 = [x_1 : x_2], x_1 = x_3\,x_3, x_2 = \texttt{nil}, x_3 = \backslash x.x, x_4 = x_3\#x_0\ \texttt{in}\ x_4]\!]\,p_0\,\kappa_1\,s_1 =$

$\kappa_1 = \lambda\varepsilon.\lambda s.\kappa_0\,\varepsilon\,\langle\rho \oplus \{main \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_1 = \langle\rho_0 \oplus \{main \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_0\rangle$

$\mathcal{E}\,[\![x_9]\!]\,p_0\,\kappa_2\,s_2 =$

mínima	máxima
$\kappa_2 = \kappa_1$	$\kappa_2 = \lambda\varepsilon.\lambda s.\bigcup_{s_f \in S_f} \kappa_1\,\varepsilon\,s_f$ $S_f = \mathsf{mforceFV}\,\{x_9\}\,s$

$s_2 = \langle\rho_1 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![[x_6 : x_7]]\!]\,\rangle, x_6 \mapsto \langle p_0, \mathcal{E}\,[\![x_8\,x_8]\!]\,\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![\texttt{nil}]\!]\,\rangle,$
$x_8 \mapsto \langle p_0, \mathcal{E}\,[\![\backslash x.x]\!]\,\rangle, x_9 \mapsto \langle p_0, \mathcal{E}\,[\![x_8\#x_5]\!]\,\rangle\}, \mathrm{sch}_1\rangle$

$\mathsf{force}\ x_9\,\kappa_2\,s_2 =$

$\mathcal{E}\,[\![x_8\#x_5]\!]\,p_0\,\kappa_3\,s_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda x.\kappa_2\,\varepsilon\,\langle\rho \oplus \{x_9 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_3 = \langle\rho_2 \oplus \{x_9 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_2\rangle$

$\mathsf{forceFV}\ x_8\,\kappa_4\,s_3 =$

$\lambda\langle\alpha, I\rangle.\lambda s.\mathsf{forceFV}\ o\,\kappa_5\,s'$

Simplificamos la continuación de expresión a la vista de la clausura ligada a x_8.

$s' = s \oplus \langle\{o \mapsto \langle p_1, (\alpha\ i)\rangle, i \mapsto \langle p_0, \mathcal{E}\,[\![x_5]\!]\,\rangle\}, \{\langle p_0, i, <>, p_1\rangle, \langle p_1, o, <>, p_0\rangle\}\rangle$

mínima	máxima
$\kappa_5 = \kappa_3$	$\kappa_5 = \lambda\varepsilon'.\lambda s''.\bigcup_{s_o \in S_o} \kappa_3\,\varepsilon\,s_o$ $S_i = \mathsf{forceFV}\ i\,(\mathsf{kstr}\ i)\,s''$ $S_8 = \bigcup_{s_i \in S_i} \mathsf{forceFV}\ o\,(\mathsf{kstr}\ o)\,s_i$

$\mathsf{forceS}\ x_8\,\kappa_6\,s_3 =$

$\kappa_6 =$ Se concretará siguiendo el primer caso de κ' de $\mathsf{forceFV}$

$\mathcal{E}\,[\![\backslash x.x]\!]\,p_0\,\kappa_7\,s_4 =$

$\kappa_7 = \lambda\varepsilon.\lambda s.\kappa_6\,\varepsilon\,\langle\rho \oplus \{x_8 \mapsto \varepsilon\}, \mathrm{sch}\rangle$

$s_4 = \langle\rho_3 \oplus \{x_8 \mapsto \mathrm{not\_ready}\}, \mathrm{sch}_3\rangle$

$\kappa_7\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_4 =$

$\kappa_6\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_5 =$

$\bigcup_{s_f \in S_f} \kappa_4\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s'_f =$

$S_f = \mathsf{mforceFV}\,\emptyset\,s_5 = \{s_5\}$

$s'_f = s_5$

$\kappa_4\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_5 =$

$\mathsf{forceFV}\ o\,\kappa_5\,s_6 =$

$s_6 = s_5 \oplus \langle\{o \mapsto \langle p_1, \mathcal{E}\,[\![i]\!]\,\rangle, i \mapsto \langle p_0, \mathcal{E}\,[\![x_5\rangle\}\}, \{\langle p_0, i, <>, p_1\rangle, \langle p_1, o, <>, p_0\rangle\}\rangle$

$\mathsf{forceS}\ o\,\kappa_8\,s_6 =$

$\kappa_8 =$ Se concretará siguiendo el tercer caso de κ' de $\mathsf{forceFV}$

$\mathcal{E}\,[\![i]\!]\,p_1\,\kappa_9\,s_7 =$

$\kappa_9 = \lambda\varepsilon.\lambda s.\kappa_8\,\varepsilon\,\langle\{\rho \oplus \{o \mapsto \varepsilon\}\}\rangle$

$s_7 = \langle\rho_6 \oplus \{o \mapsto \text{not\_ready}\}, \text{sdh}_6\rangle$

$\mathsf{force}\,i\,\kappa_9\,s_7 =$

$\mathsf{forceFV}\,i\,\kappa_9\,s_7 =$

$\mathsf{forceS}\;\kappa_{10}\,s_7 =$

$\kappa_{10} =$ Se concretará siguiendo el tercer caso de κ' de $\mathsf{forceFV}$

$\mathcal{E}\,[\![x_5]\!]\,p_0\,\kappa_{11}\,s_8 =$

$\kappa_{11} = \lambda\varepsilon.\lambda s.\kappa_{10}\,\varepsilon\,\langle\rho \oplus \{i \mapsto \varepsilon\}, \text{sdh}\rangle$

$s_8 = \langle\rho_7 \oplus \{i \mapsto \text{not\_ready}\}, \text{sdh}_7\rangle$

$\mathsf{force}\,x_5\,\kappa_1\,1\,s_8 =$

$\mathsf{forceS}\,x_5\,\kappa_1\,1\,s_8 =$

$\mathcal{E}\,[\![[x_6 : x_7]]\!]\,p_0\,\kappa_{12}\,s_9 =$

$\kappa_1 2 = \lambda\varepsilon.\lambda s.\kappa_{11}\,\varepsilon\,\langle\rho \oplus \{x_5 \mapsto \varepsilon\}, \text{sdh}\rangle$

$s_9 = \langle\rho_8 \oplus \{x_5 \mapsto \mathit{not\_ready}\}, \text{sdh}_8\rangle$

$\kappa_{12}\,\langle x_6, x_7\rangle\,s_9 =$

$\kappa_{11}\,\langle x_6, x_7\rangle\,s_{10} =$

$s_{10} = \langle\rho_8 \oplus \{x_5 \mapsto \langle x_6, x_7\rangle\}, \text{sdh}_8\rangle$

$\kappa_{10}\,\langle x_6, x_7\rangle\,s_{11} =$

$s_{11} = \langle\rho_7 \oplus \{x_5 \mapsto \langle x_6, x_7\rangle, i \mapsto \langle x_6, x_7\rangle\}, \text{sdh}_7\rangle$

$\bigcup_{s_h \in S_h} \kappa_9\,\langle x_6, c_0\rangle\,s_h = (*)$

$s'_h = s_h \oplus \langle\{c_0 \mapsto \langle p_0, \mathcal{E}\,[\![x_7]\!]\,\rangle, i \mapsto \langle x_6, c_0\rangle\}, \{\langle p_0, c_0, <\text{cv}_h>, p_1\rangle\}\rangle$

cv_h se concretará más adelante.

$S_h = \mathsf{forceFV}\,x_6\,id_\kappa\,s_{11} =$

$\mathsf{forceS}\,x_6\,\kappa_{13}\,s_{11} =$

$\kappa_{13} =$ Se concretará siguiendo el primer caso de κ' de $\mathsf{forceFV}$

$\mathcal{E}\,[\![x_8\;x_8]\!]\,p_0\,\kappa_{14}\,s_{12} =$

$\kappa_{14} = \lambda\varepsilon.\lambda s.\kappa_{13}\,\varepsilon\,\langle\rho \oplus \{x_6 \mapsto \varepsilon\}, \text{sdh}\rangle$

$s_{12} = \langle\rho_{11} \oplus \{x_6 \mapsto \text{not\_ready}\}, \text{sdh}_{11}\rangle$

$\mathcal{E}\,[\![x_8]\!]\,p_0\,\kappa_{15}\,s_{12} =$

$\kappa_{15} = \lambda\langle\alpha, I\rangle.\lambda s.(\alpha\,x_8)\,p_0\,\kappa_{14}\,s$

$\mathsf{force}\,x_8\,\kappa_{15}\,s_{12} =$

$\mathsf{forceS}\,x_8\,\kappa_{15}\,s_{12} =$

$\kappa_{15}\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_{12} =$

$\mathcal{E}\,[\![x_8]\!]\,p_0\,\kappa_{14}\,s_{12} =$

$\mathsf{force}\,x_8\,\kappa_{14}\,s_{12} =$

$\mathsf{forceS}\,x_8\,\kappa_{14}\,s_{12} =$

$\kappa_{14}\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_{12} =$

$\kappa_{13}\,\langle\lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\,s_{13} =$

$s_{13} = \langle \rho_{12} \oplus \{x_6 \mapsto \langle x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\}, \mathrm{sch12}\rangle$

$\bigcup_{s_f \in S_f} id_\kappa\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\, s'_f =$

$S_f = \mathsf{mforceFV}\,\emptyset\, s_{13} = \{s_{13}\}$

$s'_f = s_{13}$

$id_\kappa\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\, s_{13} =$

$\{s_{13}\}$

$s'_h = s_{13} \oplus \langle \{c_0 \mapsto \langle p_0, \mathcal{E}\,[\![x_7]\!]\,\rangle, i \mapsto \langle x_6, c_0\rangle\}, \{\langle p_0, c_0, <\lambda x.\mathcal{E}\,[\![x]\!]>, p_1\rangle\}\rangle = s_{14}$

$(*) = \kappa_9\,\langle x_6, c_0\rangle\, s_{14} =$

$\kappa_8\,\langle x_6, c_0\rangle\, s_{15} =$

$s_{15} = \langle \rho_{14} \oplus \{o \mapsto \langle x_6, c_0\rangle\}, \mathrm{sch}_{14}\}$

$\bigcup_{s_h \in S_h} \kappa_5\,\langle x_6, c_1\rangle\, s'_h =$

$s'_h = s_h \oplus \langle \{c_1 \mapsto \langle p_1, \mathcal{E}\,[\![c_0]\!]\,\rangle, o \mapsto \langle x_6, c_1\rangle\}, \{\langle p_1, c_1, <\mathrm{cv}_h>, p_0\rangle\}\rangle$

$S_h = \mathsf{forceFV}\, x_6\, id_\kappa\, s_{15} =$

$\mathsf{forceS}\, x_6\, \kappa_{16}\, s_{15} =$

$\kappa_{16} = $ Se concretará siguiendo el primer caso de κ' de $\mathsf{forceFV}$

$\kappa_{16}\,\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\, s_{15} =$

$\bigcup_{s_f \in S_f} id_\kappa\langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\, s'_f =$

$S_f = \mathsf{mforceFV}\,\emptyset\, s_{17} = \{s_{15}\}$

$s'_f = s_{15}$

$\{s_{15}\}$

$s'_h = s_{15} \oplus \langle \{c_1 \mapsto \langle p_1, \mathcal{E}\,[\![c_0]\!]\,\rangle, o \mapsto \langle x_6, c_1\rangle\}, \{\langle p_1, c_1, <\lambda x.\mathcal{E}\,[\![x]\!]>, p_0\rangle\}\rangle = s_{16}$

$\kappa_5\,\langle x_6, c_1\rangle\, s_{16} =$

Semántica mínima

$\kappa_3\,\langle x_6, c_1\rangle\, s_{16} =$

$\kappa_2\,\langle x_6, c_1\rangle\, s_{17} =$

$s_{17} = \langle \rho_{16} \oplus \{x_9 \mapsto \langle x_6, c_1\rangle\}, \mathrm{sch}_{16}\rangle$

$\kappa_0\,\langle x_6, c_1\rangle\, s_{18} =$

$\{s_{18}\}$

$\{\langle \{x_5 \mapsto \langle x_6, x_7\rangle, x_6 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![\mathtt{nil}]\!]\,\rangle, x_8 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle,$

$main \mapsto \langle x_6, c_1\rangle, c_0 \mapsto \langle p_0, \mathcal{E}\,[\![x_7]\!]\,\rangle, c_1 \mapsto \langle p_1, \mathcal{E}\,[\![c_0]\!]\,\rangle, o \mapsto \langle x_6, c_1\rangle, i \mapsto \langle x_6, c_0\rangle, \},$

$\{\langle p_0, c_0, <\lambda x.\mathcal{E}\,[\![x]\!]>, p_1\rangle, \langle p_1, c_1, <\lambda x.\mathcal{E}\,[\![x]\!]>, p_0\rangle\}\rangle\}$

Semántica máxima

$\bigcup_{s_o \in S_o} \kappa_3\,\langle x_6, c_1\rangle\, s_o =$

$S_o = \bigcup_{s_i \in S_i} \mathsf{forceFV}\, o\,(\mathsf{kstr}\ o)\, s_i$

$S_i = \mathsf{mforceFV}\, i\,(\mathsf{kstr}\ i)\, s_{18}$

El forzado de i (que ya no es de comunicación) lleva a forzar x_6, ya evaluada anteriormente, y c_o. Esta última sí es de comunicación, por lo que se procede a incorporar el resultado de su evaluación

al conjunto de cadenas del canal correspondiente. La evaluación da lugar a nil, por lo que el *stream* se cierra.

El forzado de o (que tampoco es de comunicación) lleva a forzar x_6, ya evaluada anteriormente, y c_1. Esta última sí es de comunicación, por lo que se procede a incorporar el resultado de su evaluación (calculada ya cuando se procedió con i) a la secuencia del correspondiente canal. De nuevo, la evaluación da lugar a nil, por lo que el *stream* se cierra.

El conjunto de estados final es:

$$\{(\{x_5 \mapsto \langle x_6, x_7\rangle, x_6 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \mathsf{nil}, x_8 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle,$$
$$main \mapsto \langle x_6, c_1\rangle, c_0 \mapsto \mathsf{nil}, c_1 \mapsto \mathsf{nil}, o \mapsto \langle x_6, c_1\rangle, i \mapsto \langle x_6, c_0\rangle, \},$$
$$\{\langle p_0, \text{closed}, <\lambda x.\mathcal{E}\,[\![x]\!]\,, \mathsf{nil}>, p_1\rangle, \quad \langle p_1, \text{closed}, <\lambda x.\mathcal{E}\,[\![x]\!]\,, \mathsf{nil}>, p_0\rangle\}\rangle\}$$

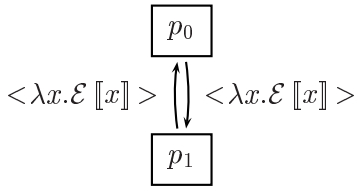
A la vista de los estados finales de las semánticas mínima y máxima llegamos a las siguientes conclusiones:

- Los canales de comunicación han sido *streams*.
- En el caso de la semántica mínima los canales no han sido cerrados, pues sólo se ha demandado la cabeza.
- En cambio, en la máxima han sido evaluados completamente y cerrados (valor closed).
- Evidentemente, los valores asociados a *main* en ambos casos coinciden.

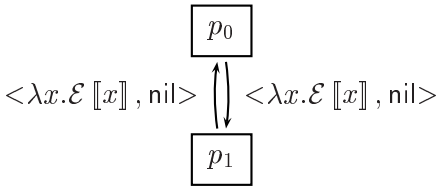
Por otra parte, la semántica máxima ha llevado a evaluar todas las variables.

Gráficamente, las topologías de procesos creados, han sido las siguientes:

Conjunto de canales - Mínima



Conjunto de canales - Máxima



□

Comparando entre sí los Ejemplos 8.8 y 8.9, se puede observar cómo cambia el comportamiento de una lista dependiendo de a qué clase de variable se encuentre asociada. En el primer ejemplo veíamos que la lista era demandada, pero se evaluaba sólo hasta llegar a *whnf*, es decir, ni la cabeza ni la cola eran evaluadas. En el segundo ejemplo, tenemos dos evaluaciones distintas para la misma lista, correspondientes a las semánticas mínima y máxima. Lo común en ambos casos es que, ante la demanda de evaluación de la lista, no solamente se demanda el constructor de ésta, sino que además se evalúa la cabeza. El motivo de esta evaluación es que la lista ya no es tal, sino un *stream*. Y la diferencia entre la semántica mínima y la máxima es que, como en la primera no se produce mayor demanda sobre el *stream*, la cola de éste no se evalúa; sin embargo, en la máxima el *stream* se evalúa al completo, comunicándose todos los valores que lo componen.

CAPÍTULO 9

Aplicando el modelo de continuaciones a otros lenguajes

> No basta tener buen ingenio; lo principal es aplicarlo bien.
>
> René Descartes

Como ya se explicó en el Capítulo 2, existen diversas formas de introducir paralelismo en los lenguajes funcionales. En el Capítulo 8 hemos definido un modelo de continuaciones para Jauja, cuyo paralelismo es explícito. En el presente capítulo emplearemos la semántica denotacional de continuaciones para definir el significado de representantes de las otras dos pautas para introducir paralelismo en los lenguajes funcionales: el paralelismo implícito y el paralelismo semi-explícito.

Así pues, el *leitmotiv* de este capítulo es posibilitar la comparación de los tres modos de introducción de paralelismo en los lenguajes funcionales, para lo que hemos seleccionado los lenguajes Eden (cuyo núcleo es Jauja), GpH y pH. Recordemos que GpH [THM+96] se enmarcaba en el paradigma semi-explícito, a cuya definición operacional dedicamos parte del Capítulo 5. Por su parte, el lenguaje pH [NA01] fue diseñado para explotar el paralelismo implícito. La elección de estos lenguajes y no otros como Facile [GMP90] o CML [Rep92] no es arbitraria, sino que se debe a la relación subyacente entre ellos; a saber, Eden, GpH y pH son todos dialectos del lenguaje funcional perezoso por excelencia: Haskell [PH99, Pey03]. Al definir la semántica de las construcciones básicas de Jauja y sendos núcleos de GpH y pH en un marco denotational común de continuaciones podremos realizar un estudio comparativo de estos tres enfoques de paralelismo.

Comencemos con la semántica de continuaciones para GpH.

9.1. Semántica denotacional para GpH

En el Capítulo 5 ya se detalló la semántica operacional para GpH propuesta en [BKT00]. Junto a ella, en dicho artículo se incluía una semántica denotacional directa. Sin embargo, esta semántica no expresaba de manera precisa el orden de evaluación perezoso que subyace en GpH.

El núcleo sintáctico de GpH que vamos a usar aquí es básicamente el mismo que fue considerado en su momento en [BKT00] y que quedó recogido en la Figura 5.5 del Capítulo 5. Al igual que hemos hecho para Jauja, tanto en la semántica operacional como en la denotacional, usaremos una sintaxis restringida, que si se compara con la que se empleó en [BKT00], incluye aún más sustituciones de expresiones por variables para lograr la semántica de pereza completa descrita en la Sección 4.3. La sintaxis restringida de GpH-core queda recogida en la Figura 9.1.

Sintaxis Restringida		
E ::=	x	variable
|	$\backslash x.E$	λ-abstracción
|	$x_1\ x_2$	aplicación
|	`let` $\{x_i = E_i\}_{i=1}^{n}$ `in` x	declaración local
|	x_1 `'seq'` x_2	composición secuencial
|	x_1 `'par'` x_2	composición paralela

Figura 9.1: GpH-core restringido

Se observa que las dos subexpresiones de la aplicación funcional han de ser variables, como ya hicimos en Jauja. Del mismo modo, el cuerpo de la declaración local de variables es también una variable. En cuanto a las construcciones propias de GpH, también las dos subexpresiones de la composición son sustituidas por variables.

9.1.1. Dominios semánticos

Los dominios semánticos que se precisan para construir el modelo denotacional de continuaciones para GpH están expuestos en la Figura 9.2.

La explicación de estos dominios se va a realizar mediante una comparación de los mismos con los expuestos para Jauja básico en la Figura 8.1. En primer lugar, en GpH no existe no-determinismo, por lo que las continuaciones tendrán que transformar un estado en otro, y no en un conjunto de estados, como sucedía en Jauja. De igual manera, como el paralelismo que se tiene en GpH-core es de hebras y no de procesos, las comunicaciones entre éstos no tienen lugar. Por ello, el dominio que se tenía en Jauja para representar canales de comunicación no va a aparecer aquí. Estos dos cambios provocan que en los

	Cont	=	**Env** → **Env**	continuaciones
$\kappa \in$	**ECont**	=	**EVal** → **Cont**	continuaciones de expresión
$\rho \in$	**Env**	=	**Ide** → (**Val** + {undefined})	entornos
$v \in$	**Val**	=	**EVal** + **Clo** + {not_ready}	valores
$\varepsilon \in$	**EVal**	=	**Abs**	valores de expresión
$\alpha \in$	**Abs**	=	**Ide** → **Clo**	valores de abstracción
$\nu \in$	**Clo**	=	**ECont** → **Cont**	clausuras

Figura 9.2: GPH-CORE: dominios semánticos

estados solamente aparezca la componente entorno, y que una continuación transforme un entorno en otro entorno, **Cont** = **Env** → **Env**. Los entornos sí conservan su forma general, **Env** = **Ide** → **Val**; sin embargo, los valores van a tener formas distintas. Debido también al hecho de que en GPH no existe trasiego de comunicaciones, no se va a tener que "copiar" ligaduras de un *heap* a otro, por lo que desaparece la necesidad de conservar las variables libres junto con el valor denotacional de una abstracción. Y por no existir procesos, las clausuras se simplifican, no teniendo que suministrar el identificador de proceso en el que se evalúa una clausura; tampoco se asocia con una clausura su proceso generador. Con todo esto, el dominio de valores **Val** queda simplificado a **EVal** + **Clo** + {not_ready}, mientras que el de valores de expresión es sinónimo del de abstracciones, que siguen siendo las funciones en **Ide** → **Clo**, mientras que las clausuras vienen dadas por **ECont** → **Cont**. Por último, las continuaciones de expresión mantienen el mismo tipo que en Jauja y, como veremos más adelante, también lo conservarán en el caso de pH.

9.1.2. Función de evaluación

El tipo de la función de evaluación $\mathcal{E}$ difiere ligeramente del de la misma función para Jauja, ya que la ausencia de procesos en GPH simplifica la signatura de $\mathcal{E}$:

$$\mathcal{E} :: \mathbf{Exp} \to \mathbf{ECont} \to \mathbf{Cont}.$$

Esta función, que explicamos seguidamente, queda definida en la Figura 9.3.

La evaluación de una *variable* es similar a la que se realizaba para Jauja en todas las versiones incluidas en el Capítulo 8. También hay que forzar la evaluación de la variable, para lo cual empleamos la función auxiliar force, descrita en la Sección 9.1.3.

La obtención del valor denotacional de una *abstracción funcional* es muy similar a la que se realizaba en Jauja: se tiene que construir una abstracción denotacional, es decir, una función que tome un identificador y devuelva una clausura: $\lambda x.\mathcal{E}\,[\![E]\!]$ es el valor de expresión que recibe la continuación de expresión κ.

La *aplicación funcional perezosa*, comparada con las detalladas para el caso de Jauja (Figuras 8.2 y 8.6), es también más sencilla, pues los identificadores correspondientes a

$$\mathcal{E}\,[\![x]\!]\,\kappa = \mathsf{force}\,x\,\kappa$$

$$\mathcal{E}\,[\![\backslash x.E]\!]\,\kappa = \kappa(\lambda x.\mathcal{E}\,[\![E]\!]\,)$$

$$\mathcal{E}\,[\![x_1\;x_2]\!]\,\kappa = \mathcal{E}\,[\![x_1]\!]\,\kappa' \quad \texttt{donde}\ \kappa' = \lambda\varepsilon.\lambda\rho.\varepsilon\,x_2\,\kappa\,\rho$$

$$\mathcal{E}\,[\![x_1\ \texttt{'seq'}\ x_2]\!]\,\kappa = \mathcal{E}\,[\![x_1]\!]\,\kappa' \quad \texttt{donde}\ \kappa' = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![x_2]\!]\,\kappa\,\rho$$

$$\mathcal{E}\,[\![x_1\ \texttt{'par'}\ x_2]\!]\,\kappa = \mathcal{E}\,[\![x_2]\!]\,\kappa' \quad \texttt{donde}\ \kappa' = \lambda\varepsilon.\lambda\rho.\kappa\,\varepsilon\,\rho_{par},\ \rho_{par} = \mathsf{par}\;x_1\,\rho$$

$$\mathcal{E}\,[\![\texttt{let}\ \{x_i = E_i\}_n\ \texttt{in}\ x]\!]\,\kappa = \lambda\rho.\mathcal{E}\,[\![x[y_1/x_1,\ldots,y_n/x_n]]\!]\,\kappa\,\rho' \quad \texttt{donde}\ \{y_1,\ldots,y_n\} = \mathsf{newIde}\,n\,\rho,\ \rho' = \rho \oplus \{y_i \mapsto \mathcal{E}\,[\![E_i[y_1/x_1,\ldots,y_n/x_n]]\!]\ \mid 1 \le i \le n\}$$

Figura 9.3: GPH-CORE: función de evaluación

las variables libres son innecesarios. En definitiva, se evalúa la componente que conformará la abstracción, y la continuación de expresión *ad hoc* κ' toma el valor de expresión obtenido —que solamente puede ser una abstracción— y lo aplica a la variable argumento, siendo la continuación de expresión de esta aplicación la continuación de expresión inicial, κ. De este modo se modeliza el orden y modo de evaluación deseado: evaluación perezosa (*call-by-need* con compartición).

La *declaración local de variables* es mucho más simple que en las semánticas para Jauja, pues GPH no crea procesos. Únicamente hay que incluir en el entorno cada variable —que introduciremos nueva para evitar choques de nombres— con su clausura correspondiente.

En cuanto a la *composición secuencial*, se marca el orden de evaluación con ayuda de la continuación. Sin embargo, no se devuelve como resultado de la composición el valor obtenido a partir de x_1, sino el que se produce a partir de x_2, por lo que la continuación de expresión inicial κ prosigue a la evaluación de x_2. Se observa que este esquema es similar al dado en el Capítulo 7 para la composición secuencial de instrucciones cuando en el lenguaje no existían ni saltos ni etiquetas.

Finalmente, la semántica de la *composición paralela* también tiene que definirse devolviendo el valor obtenido para x_2. Sin embargo, en el estado tienen que reflejarse los potenciales cambios debidos a la evaluación en paralelo de x_1. Por ello, el contexto —que junto con la nueva continuación de expresión se pasa a la evaluación de x_2— se encarga de forzar la evaluación de x_1, si hubiera suficientes recursos disponibles. Esta última decisión la lleva a cabo la función auxiliar par, que, junto con la función force, se incluye y explica a continuación.

9.1.3. Funciones semánticas auxiliares

En el caso de GPH hemos podido observar que las funciones semánticas auxiliares se reducen a force y par, ambas definidas en la Figura 9.4.

La función force es análoga —salvo por la adaptación de estados— a la presentada

```
force :: Ide → ECont → Cont
force x κ = λρ.case (ρ x) of
  ε ∈ EVal ⟶ κ ε ρ
  ν ∈ Clo ⟶ ν κ' ρ'
    donde κ' = λε''.λρ''.κ ε'' ρ'' ⊕ {x ↦ ε''}
          ρ' = ρ ⊕ {x ↦ not_ready}
  not_ready ⟶ wrong
endcase
```

```
par :: Ide → Cont
par_min x = λρ.ρ
par_max x = λρ.E [[x]] id_κ ρ
```

Figura 9.4: GPH-CORE: funciones semánticas auxiliares

para Jauja en la Figura 8.3. En cuanto a la función par, su misión es decidir si se evalúa o no en paralelo la variable que se le suministra. En el caso de la semántica mínima no se procede a evaluación alguna; sin embargo, en el caso de la semántica máxima siempre se evalúa la variable, con la continuación de expresión identidad para devolver tal cual el estado resultante. Podrían definirse diferentes versiones de esta función para recoger las múltiples posibilidades de disponibilidad de procesadores en el sistema.

Por último, debemos señalar que la continuación de expresión identidad, id_κ, se define en este caso de la forma siguiente:

$$id_\kappa :: \mathbf{EVal} \to \mathbf{Cont}$$
$$id_\kappa = \lambda\varepsilon.\lambda\rho.\rho$$

A continuación, utilizaremos la semántica que acabamos de definir para estudiar algunas propiedades de GPH.

9.1.4. Propiedades

Es posible utilizar la semántica de continuaciones para demostrar equivalencias entre las expresiones de GPH, como las indicadas en [HBTK99].

Es evidente que no se pueden demostrar directamente propiedades como x `'par'` $y \equiv x$ `'par'` $(x$ `'par'` $y)$, pues esta última expresión no se encuentra normalizada. Recojamos en la siguiente definición el modo de elegir las variables nuevas de la normalización.

Definición 9.1 Sea x una variable. Decimos que x' es una *variable fresca con respecto a* x si, dados una continuación de expresión κ y un entorno ρ, $\mathcal{E}\,[\![x]\!]\,\kappa\,(\rho \oplus \{x_i \mapsto \text{not\_ready}\})$ devuelve un entorno ρ' tal que

1. $\rho'\,x = \varepsilon_x$, o bien
2. $\rho'\,x = \text{undefined}$, o bien
3. $\rho'\,x = \text{not\_ready}$, no siendo esta asociación provocada por el hecho de estar x_i ligada a not_ready.

Obsérvese que de estas condiciones se deduce que si $\rho\, x_i = \nu$, entonces x_i no va a ser forzada, y que si $\rho\, x_i = \varepsilon_i$, entonces x_i no va a ser necesaria en la evaluación de x.

□

Este concepto ya fue formalizado para Jauja en la Definición 8.2.

En primer lugar demostramos la asociatividad de la construcción 'seq':

Proposición 9.1 *Sea ρ_0 un entorno, x_2, x_3 variables frescas con respecto a las variables x e y. Sea κ_0 la continuación de expresión $\lambda\varepsilon.\lambda\rho.\{x \mapsto \rho\, x, y \mapsto \rho\, y\}$, entonces se tiene:*

$$\mathcal{E}\,[\![\texttt{let } x_0 = y \texttt{ 'seq' } z, x_1 = x \texttt{ 'seq' } x_0 \texttt{ in } x_1]\!]\,\kappa_0\,\rho_0 = \mathcal{E}\,[\![\texttt{let } x_0 = x \texttt{ 'seq' } y, x_1 = x_0 \texttt{ 'seq' } z \texttt{ in } x_1]\!]\,\kappa_0\,\rho_0$$

Demostración P.9.1
Para demostrarlo calculamos los valores de ambas expresiones. Obsérvese que, al no aparecer el operador paralelo, nuestros cálculos serán independientes de si la semántica tratada es la mínima o la máxima.

$\mathcal{E}\,[\![\texttt{let } x_0 = y \texttt{ 'seq' } z, x_1 = x \texttt{ 'seq' } x_0 \texttt{ in } x_1]\!]\,\kappa_0\,\rho_0 =$
$\mathcal{E}\,[\![x_3]\!]\,\kappa_1\,\rho_1 =$
$\kappa_1 = \kappa_0$
$\rho_1 = \rho_0 \oplus \{x_2 \mapsto \mathcal{E}\,[\![y \texttt{ 'seq' } z]\!], x_3 \mapsto \mathcal{E}\,[\![x \texttt{ 'seq' } x_2]\!]\}$
force $x_3\,\kappa_1\,\rho_1 =$
$\mathcal{E}\,[\![x \texttt{ 'seq' } x_2]\!]\,\kappa_2\,\rho_2 =$
$\kappa_2 = \lambda\varepsilon.\lambda\rho.\kappa_1\,\varepsilon\,(\rho \oplus \{x_3 \mapsto \varepsilon\})$
$\rho_2 = \rho_1 \oplus \{x_3 \mapsto \text{not\_ready}\}$
$\mathcal{E}\,[\![x]\!]\,\kappa_3\,\rho_3 =$
$\kappa_3 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![x_2]\!]\,\kappa_2\,\rho$
$\rho_3 = \rho_2$
Esta evaluación dará lugar a un entorno ρ', en el que x estará asociada o a not_ready o a un valor ε_x. Consideremos ahora el segundo caso. Además, ρ' debe contener $\{x_3 \mapsto \text{not\_ready}, x_2 \mapsto \mathcal{E}\,[\![y \texttt{ 'seq' } z]\!]\}$.
$\kappa_3\,\varepsilon_x\,\rho' =$
$\mathcal{E}\,[\![x_2]\!]\,\kappa_2\,\rho' =$
force $x_2\,\kappa_2\,\rho'$
$\mathcal{E}\,[\![y \texttt{ 'seq' } z]\!]\,\kappa_4\,\rho_4$
$\kappa_4 = \lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,(\rho \oplus \{x_2 \mapsto \varepsilon\})$
$\rho_4 = \rho' \oplus \{x_2 \mapsto \text{not\_ready}\}$
$\mathcal{E}\,[\![y]\!]\,\kappa_5\,\rho_4$
$\kappa_5 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![z]\!]\,\kappa_4\,\rho$

$\mathcal{E}\,[\![\texttt{let } x_0 = x \texttt{ 'seq' } y, x_1 = x_0 \texttt{ 'seq' } z \texttt{ in } x_1]\!]\,\kappa_0\,\rho_0 =$
$\mathcal{E}\,[\![x_3]\!]\,\kappa_1\,\rho_1 =$
$\kappa_1 = \kappa_0$
$\rho_1 = \rho_0 \oplus \{x_2 \mapsto \mathcal{E}\,[\![x \texttt{ 'seq' } y]\!], x_3 \mapsto \mathcal{E}\,[\![x_2 \texttt{ 'seq' } z]\!]\}$
force $x_3\,\kappa_1\,\rho_1 =$
$\mathcal{E}\,[\![x_2 \texttt{ 'seq' } z]\!]\,\kappa_2\,\rho_2 =$
$\kappa_2 = \lambda\varepsilon.\lambda\rho.\kappa_1\,\varepsilon\,(\rho \oplus \{x_3 \mapsto \varepsilon\})$
$\rho_2 = \rho_1 \oplus \{x_3 \mapsto \text{not\_ready}\}$
$\mathcal{E}\,[\![x_2]\!]\,\kappa_3\,\rho_3$
$\kappa_3 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![z]\!]\,\kappa_2\,\rho$
$\rho_3 = \rho_2$
force $x_2\,\kappa_3\,\rho_3$
$\mathcal{E}\,[\![x \texttt{ 'seq' } y]\!]\,\kappa_4\,\rho_4$
$\kappa_4 = \lambda\varepsilon.\lambda\rho.\kappa_3\,\varepsilon\,(\rho \oplus \{x_2 \mapsto \varepsilon\})$
$\rho_4 = \rho_3 \oplus \{x_2 \mapsto \text{not\_ready}\}$
$\mathcal{E}\,[\![x]\!]\,\kappa_5\,\rho_4$
$\kappa_5 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![y]\!]\,\kappa_4\,\rho$

Observamos que en ambos casos se evalúan las tres variables x, y y z:

- La variable x se evalúa en el entorno inicial, modificado solamente por variables frescas con respecto a x, con lo que el valor obtenido es el mismo en los dos casos.
- La variable y se evalúa en el entorno que resulta de evaluar x. En ambos casos, estos dos entornos serán iguales salvo por los valores asociados a las variables frescas con respecto a x y a y. En consecuencia, los valores para y coinciden.
- Finalmente, la evaluación de z se realiza en el entorno devuelto por la evaluación de y. Razonando análogamente, se deduce que el valor de z también es el mismo en ambos casos.

En conclusión, los entornos finales son el mismo en los dos casos.

c.q.d.

Veamos ahora la idempotencia del operador de paralelo con respecto al primer operador.

Proposición 9.2 *Sea ρ_0 un entorno, x_1, x_2 y x_3 variables frescas con respecto a la variables x e y, y κ_0 la continuación de expresión $\lambda\varepsilon.\lambda\rho.\{x \mapsto \rho\, x, y \mapsto \rho\, y\}$. Entonces se tiene:*

$$\mathcal{E}\,[\![x\ \texttt{'par'}\ y]\!]\,\kappa_0\,\rho_0 = \mathcal{E}\,[\![\texttt{let}\ x_0 = x\ \texttt{'par'}\ y, x_1 = x\ \texttt{'par'}\ x_0\ \texttt{in}\ x_1]\!]\,\kappa_0\,\rho_0$$

Demostración P.9.2
Calculemos los valores de ambas expresiones, empezando por la semántica mínima:

$\mathcal{E}\,[\![x\ \texttt{'par'}\ y]\!]\,\kappa_0\,\rho_0 =$
$\mathcal{E}\,[\![y]\!]\,\kappa_1\,\rho_1 =$
$\quad \kappa_1 = \lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,\rho_{par}$
$\quad \rho_1 = \rho_0$
$\quad \rho_{par} = \mathsf{par}\ x\,\rho = \rho$
$\mathcal{E}\,[\![y]\!]\,\kappa_0\,\rho_0$

$\mathcal{E}\,[\![\texttt{let}\ x_0 = x\ \texttt{'par'}\ y, x_1 = x\ \texttt{'par'}\ x_0\ \texttt{in}\ x_1]\!]\,\kappa_0\,\rho_0 =$
$\mathcal{E}\,[\![x_3]\!]\,\kappa_1\,\rho_1 =$
$\quad \kappa_1 = \kappa_0$
$\quad \rho_1 = \rho_0 \oplus \{x_2 \mapsto \mathcal{E}\,[\![x\ \texttt{'par'}\ y]\!], x_3 \mapsto \mathcal{E}\,[\![x\ \texttt{'par'}\ x_2]\!]\}$
$\mathsf{force}\ x_3\,\kappa_1\,\rho_1 =$
$\mathcal{E}\,[\![x\ \texttt{'par'}\ x_2]\!]\,\kappa_2\,\rho_2 =$
$\quad \kappa_2 = \lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,(\rho \oplus \{x_3 \mapsto \varepsilon\})$
$\quad \rho_2 = \rho_1 \oplus \{x_3 \mapsto \text{not\_ready}\}$
$\mathcal{E}\,[\![x_2]\!]\,\kappa_3\,\rho_3$
$\quad \kappa_3 = \lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,\rho_{par}$
$\quad \rho_3 = \rho_2$
$\quad \rho_{par} = \mathsf{par}\ x\,\rho = \rho$
$\mathsf{force}\ x_2\,\kappa_3\,\rho_3$
$\mathcal{E}\,[\![x\ \texttt{'par'}\ y]\!]\,\kappa_4\,\rho_4$
$\quad \kappa_4 = \lambda\varepsilon.\lambda\rho.\kappa_3\,\varepsilon\,(\rho \oplus \{x_2 \mapsto \varepsilon\})$
$\quad \rho_4 = \rho_3 \oplus \{x_2 \mapsto \text{not\_ready}\}$
$\mathcal{E}\,[\![y]\!]\,\kappa_5\,\rho_4 =$
$\quad \kappa_5 = \lambda\varepsilon.\lambda\rho.\kappa_4\,\varepsilon\,\rho_{par_2}$
$\quad \rho_{par_2} = \mathsf{par}\ x\,\rho = \rho$
$\mathcal{E}\,[\![y]\!]\,\kappa_0\,(\rho_0 \oplus \{x_2 \mapsto \text{not\_ready}, x_3 \mapsto \text{not\_ready}\})$

Los valores son iguales salvo por el hecho de que en el primer caso el entorno es ρ_0, y en el segundo éste se extiende modificando los undefined de x_2 y x_3 por not_ready. Pero como x_2 y x_3 son frescas con respecto a y, se deduce que en los dos entornos finales se tendrán los mismos valores asociados a x y a y.

Estudiemos ahora el caso de la semántica máxima. Las evaluaciones son parecidas salvo que:

$$\rho_{par} = \mathsf{par}\ x\ \rho = \mathcal{E}\ [\![x]\!]\ id_\kappa\ \rho \text{ y } \rho_{par_2} = \mathsf{par}\ x\ \rho = \mathcal{E}\ [\![x]\!]\ id_\kappa\ \rho;$$

lo que da lugar a los valores finales:

$$\begin{array}{l|l} \mathcal{E}\ [\![y]\!]\ (\lambda\varepsilon.\lambda\rho.\{x \mapsto \rho' x, y \mapsto \rho' y\})\ \rho_0 & \mathcal{E}\ [\![y]\!]\ (\lambda\varepsilon.\lambda\rho.\{x \mapsto \rho' x, y \mapsto \rho' y\}) \\ \quad \rho' = \mathcal{E}\ [\![x]\!]\ id_\kappa\ \rho & \quad\quad (\rho_0 \oplus \{x_2 \mapsto \text{not\_ready}, x_3 \mapsto \text{not\_ready}\}) \\ & \quad\quad \rho' = \mathcal{E}\ [\![x]\!]\ id_\kappa\ (\mathcal{E}\ [\![x]\!]\ id_\kappa\ \rho) = \mathcal{E}\ [\![x]\!]\ id_\kappa\ \rho \end{array}$$

De nuevo, se tiene que los dos entornos finales son iguales.

c.q.d.

La siguiente proposición establece la propiedad distributiva de la construcción `'seq'` con respecto a `'par'`.

Proposición 9.3 *Sea ρ_0 un entorno, x_0, x_1, x_2 y x_3 frescas con respecto a x, y, z, y κ_0 la continuación de expresión $\lambda\varepsilon.\lambda\rho.\{x \mapsto \rho\, x, y \mapsto \rho\, y, z \mapsto \rho\, z\}$. Entonces se tiene:*

$$\begin{array}{l} \mathcal{E}\ [\![\texttt{let } x_0 = y \texttt{ 'par' } z, x_1 = x \texttt{ 'seq' } x_0 \texttt{ in } x_1]\!]\ \kappa_0\ \rho_0 = \\ \quad \mathcal{E}\ [\![\texttt{let } x_0 = x \texttt{ 'seq' } y, x_1 = x \texttt{ 'seq' } z, x_2 = x_0 \texttt{ 'par' } x_1 \texttt{ in } x_2]\!]\ \kappa_0\ \rho_0. \end{array}$$

Demostración P.9.3
Procedamos calculando los valores de ambas partes:

$\mathcal{E}\ [\![\texttt{let } x_0 = y \texttt{ 'par' } z, x_1 = x \texttt{ 'seq' } x_0 \texttt{ in } x_1]\!]\ \kappa_0\ \rho_0 =$

$\mathcal{E}\ [\![x_3]\!]\ \kappa_1\ \rho_1 =$

$\quad \kappa_1 = \kappa_0$

$\quad \rho_1 = \rho_0 \oplus \{x_2 \mapsto \mathcal{E}\ [\![y \texttt{ 'par' } z]\!], x_3 \mapsto \mathcal{E}\ [\![x \texttt{ 'seq' } x_2]\!]\}$

$\mathsf{force}\ x_3\ \kappa_1\ \rho_1 =$

$\mathcal{E}\ [\![x \texttt{ 'seq' } x_2]\!]\ \kappa_2\ \rho_2$

$\quad \kappa_2 = \lambda\varepsilon.\lambda\rho.\kappa_1\ \varepsilon\ (\rho \oplus \{x_3 \mapsto \varepsilon\})$

$\quad \rho_2 = \rho_1 \oplus \{x_3 \mapsto \text{not\_ready}\}$

$\mathcal{E}\ [\![x]\!]\ \kappa_3\ \rho_3 =$

$\quad \kappa_3 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\ [\![x_2]\!]\ \kappa_2\ \rho$

$\quad \rho_3 = \rho_2$

El valor que resulte de evaluar x puede ser not_ready o ε_x. El primer caso lo consideramos más tarde. En el segundo, el entorno ρ' debe contener: $\{x \mapsto \varepsilon_x, x_2 \mapsto \mathcal{E}\ [\![x \texttt{ 'par' } y]\!], x_3 \mapsto \text{not\_ready}\}$

$\kappa_3\ \varepsilon_x\ \rho' =$

$\mathcal{E}\ [\![x_2]\!]\ \kappa_2\ \rho' =$

$\mathsf{force}\ x_2\ \kappa_2\ \rho' =$

$\mathcal{E}\,[\![y\ \texttt{'par'}\ z]\!]\,\kappa_4\,\rho_4$

$\kappa_4 = \lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,(\rho \oplus \{x_2 \mapsto \varepsilon\})$

$\rho_4 = \rho' \oplus \{x_2 \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![z]\!]\,\kappa_5\,\rho_5 =$

$\rho_5 = \rho_4$

$\kappa_5 = \lambda\varepsilon.\lambda\rho.\kappa_4\,\varepsilon\,\rho_{par}$

$\rho_{par} = \mathsf{par}\ y\,\rho$

Semántica mínima	Semántica máxima
$\rho_{par} = \rho$	$\rho_{par} = \mathsf{par}\ y\,\rho = \mathcal{E}\,[\![y]\!]\ id_\kappa\,\rho$
$\kappa_5 = \kappa_4$	$\kappa_5 = \lambda\varepsilon.\lambda\rho.\kappa_4\,\varepsilon\,(\mathcal{E}\,[\![y]\!]\ id_\kappa\,\rho)$
$\mathcal{E}\,[\![z]\!]\,\kappa_4\,\rho_4 =$	$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,((\mathcal{E}\,[\![y]\!]\ id_\kappa\,\rho) \oplus \{x_2 \mapsto \varepsilon, x_3 \mapsto \varepsilon\}))$
$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,(\rho \oplus \{x_2 \mapsto \varepsilon, x_3 \mapsto \varepsilon\}))$	$\quad(\rho' \oplus \{x_2 \mapsto \text{not\_ready}\})$
$\quad(\rho' \oplus \{x_2 \mapsto \text{not\_ready}\})$	
$\mathcal{E}\,[\![z]\!]\,\kappa_0\,(\rho' \oplus \{x_2 \mapsto \text{not\_ready}\})$	

$\mathcal{E}\,[\![\texttt{let}\ x_0 = x\ \texttt{'seq'}\ y, x_1 = x\ \texttt{'seq'}\ z, x_2 = x_0\ \texttt{'par'}\ x_1\ \texttt{in}\ x_2]\!]\,\kappa_0\,\rho_0 =$

$\mathcal{E}\,[\![x_5]\!]\,\kappa_1\,\rho_1 =$

$\kappa_1 = \kappa_0$

$\rho_1 = \rho_0 \oplus \{x_3 \mapsto \mathcal{E}\,[\![x\ \texttt{'seq'}\ y]\!]\,,$

$x_4 \mapsto \mathcal{E}\,[\![x\ \texttt{'seq'}\ z]\!]\,,$

$x_5 \mapsto \mathcal{E}\,[\![x_3\ \texttt{'par'}\ x_4]\!]\,\}$

$\mathsf{force}\,x_5\,\kappa_1\,\rho_1 =$

$\mathcal{E}\,[\![x_3\ \texttt{'par'}\ x_4]\!]\,\kappa_2\,\rho_2 =$

$\kappa_2 = \lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,(\rho \oplus \{x_5 \mapsto \varepsilon\})$

$\rho_2 = \rho_1 \oplus \{x_5 \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![x_4]\!]\,\kappa_3\,\rho_3$

$\kappa_3 = \lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,\rho_{par}$

$\rho_3 = \rho_2$

$\rho_{par} = \mathsf{par}\ x_3\,\rho = \rho$

$\mathsf{force}\,x_4\,\kappa_3\,\rho_3$

$\mathcal{E}\,[\![x\ \texttt{'seq'}\ z]\!]\,\kappa_4\,\rho_4$

$\kappa_4 = \lambda\varepsilon.\lambda\rho.\kappa_3\,\varepsilon\,(\rho \oplus \{x_4 \mapsto \varepsilon\})$

$\rho_4 = \rho_3 \oplus \{x_4 \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![x]\!]\,\kappa_5\,\rho_5 =$

$\kappa_5 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![z]\!]\,\kappa_4\,\rho$

$\rho_5 = \rho_4$

El valor que resulte de evaluar x puede ser not_ready o ε'_x. El primer caso lo consideramos más tarde. En el segundo, el entorno ρ' debe contener: $\{x \mapsto \varepsilon'_x, x_3 \mapsto \mathcal{E}\,[\![x\ \texttt{'seq'}\ y]\!]\,, x_4 \mapsto \text{not\_ready}, x_5 \mapsto \text{not\_ready}\}$

$\kappa_5\,\varepsilon'_x\,\rho'$

$\mathcal{E}\,[\![z]\!]\,\kappa_4\,\rho' =$

$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_3\,\varepsilon\,(\rho \oplus \{x_4 \mapsto \varepsilon\}))\,\rho' =$

Semántica mínima

$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,(\rho \oplus \{x_4 \mapsto \varepsilon\}))\,\rho' =$

$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_1\,\varepsilon\,(\rho \oplus \{x_4 \mapsto \varepsilon, x_5 \mapsto \varepsilon\}))\,\rho' =$

$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,(\rho \oplus \{x_4 \mapsto \varepsilon, x_5 \mapsto \varepsilon\}))\,\rho' =$

$\mathcal{E}\,[\![z]\!]\,\kappa_0\,\rho'$

Semántica máxima

$\rho_{par} = \mathsf{par}\ x_3\,(\rho \oplus \{x_4 \mapsto \varepsilon\}) =$

$\quad \mathcal{E}\,[\![x_3]\!]\,id_\kappa\,(\rho \oplus \{x_4 \mapsto \varepsilon\})$

$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,(\mathcal{E}\,[\![x_3]\!]\,id_\kappa\,(\rho \oplus \{x_4 \mapsto \varepsilon\}))$

$\mathcal{E}\,[\![z]\!]\,(\lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,((\mathcal{E}\,[\![x_3]\!]\,id_\kappa\,(\rho \oplus \{x_4 \mapsto \varepsilon\})) \oplus \{x_5 \mapsto \varepsilon\}))$

$\quad (\rho' \oplus \{x_2 \mapsto \text{not\_ready}\})$

El valor que resulte de evaluar z puede ser not_ready o ε_z.

El primer caso lo consideramos más tarde.

En el segundo caso, el entorno ρ'' debe contener:

$\{z \mapsto \varepsilon_z\}$ y las variables x, x_3, x_4 y x_5 ligadas

a los valores anteriores a la evaluación de z.

$\kappa_0\,\varepsilon_z\,(\mathcal{E}\,[\![x_3]\!]\,id_\kappa\,(\rho'' \oplus \{x_4 \mapsto \varepsilon\})) =$

$\quad \mathcal{E}\,[\![x_3]\!]\,id_\kappa\,(\rho_z \oplus \{x_4 \mapsto \varepsilon\})$

$\quad \mathsf{force}\ x_3\ id_\kappa\,(\rho_z \oplus \{x_4 \mapsto \varepsilon\}) =$

$\quad \mathcal{E}\,[\![x\ \texttt{'seq'}\ y]\!]\,\kappa_6\,\rho_6 =$

$\quad\quad \kappa_6 = \lambda\varepsilon.\lambda\rho.id_\kappa\,\varepsilon\,(\rho \oplus \{x_3 \mapsto \varepsilon\})$

$\quad\quad \rho_6 = \rho'' \oplus \{x_4 \mapsto \varepsilon, x_3 \mapsto \text{not\_ready}\}$

$\quad \mathcal{E}\,[\![x]\!]\,\kappa_7\,\rho_7 =$

$\quad\quad \kappa_7 = \lambda\varepsilon.\lambda\rho.\mathcal{E}\,[\![y]\!]\,\kappa_6\,\rho$

$\quad\quad \rho_7 = \rho_6$

$\quad \kappa_7\,\varepsilon'_x\,\rho_7 =$

$\quad \mathcal{E}\,[\![y]\!]\,\kappa_6\,\rho_6$

Analicemos los resultados obtenidos para la semántica mínima. Nos tenemos que detener en los valores que finalmente quedan asociados a las variables x, y, y z:

- La variable x, en el primer caso está ligada a ε_x y se ha evaluado sobre $\rho_3 = \rho_0 \oplus \{x_3 \mapsto \text{not\_ready}, x_2 \mapsto \mathcal{E}\,[\![y\ \texttt{'par'}\ z]\!]\}$, y en el segundo a ε'_x, obtenido en el entorno $\rho_5 = \rho_0 \oplus \{x_5 \mapsto \text{not\_ready}, x_5 \mapsto \text{not\_ready}, x_3 \mapsto \mathcal{E}\,[\![x\ \texttt{'seq'}\ y]\!]\}$. Como las variables x_i eran frescas con respecto a x, la evaluación en ambos casos procede sobre ρ_0, con lo que $\varepsilon_x = \varepsilon'_x$. Si la evaluación no devolviera ningún valor, sino not_ready o undefined, lo haría igual en ambos casos por el mismo motivo.

- y no es evaluada en ninguno de los dos casos, con lo que en ambos queda ligada al valor inicial.

- z se evalúa en el primer caso sobre el entorno resultado de evaluar x, y que contiene $\{x \mapsto \varepsilon_x, x_2 \mapsto \mathcal{E}\,[\![x\ \texttt{'par'}\ y]\!], x_3 \mapsto \text{not\_ready}\}$, con la modificación de x_2 a not_ready. En el segundo caso, la evaluación es también sobre el entorno obtenido tras evaluar x, y que contiene $\{x \mapsto \varepsilon'_x, x_3 \mapsto \mathcal{E}\,[\![x\ \texttt{'seq'}\ y]\!], x_4 \mapsto \text{not\_ready}, x_5 \mapsto \text{not\_ready}\}$. En ambos casos el resto del entorno es el mismo, pues se partía de ρ_0 modificado por

variables frescas, tanto con respecto a x como con respecto a z. De esto se deduce que la evaluación de z en ambos entornos producirá el mismo valor.

En conclusión, para las variables x, y y z se obtienen los mismos valores; como κ_0 solamente se queda con la parte del entorno final relativa a estos tres identificadores, los entornos resultantes de las evaluaciones de las dos expresiones iniciales son iguales en la semántica mínima.

En cuanto a la semántica máxima, los razonamientos para x y z son análogos. Detengámonos en la variable y, que en ambos casos es evaluada sobre el entorno resultado de evaluar z; los dos entornos solamente diferían en las variables que eran frescas con respecto a x, y y z. Se deduce entonces que la evaluación de y también producirá el mismo valor en ambos casos. En conclusión, los entornos resultantes de las evaluaciones de las dos expresiones iniciales son iguales en la semántica máxima.

Por último, el razonamiento en los casos donde la variable x o la y estuvieran ligadas a not_ready es análogo al expuesto.

c.q.d.

Pasamos ahora a definir una semántica denotacional de continuaciones para pH.

9.2. Semántica denotacional para pH

De pH no hemos mencionado hasta el momento más que el modo en que introduce paralelismo en Haskell. Por ello nos detendremos ahora un poco en explicar las principales características de este lenguaje funcional paralelo. Una exposición exhaustiva del mismo puede encontrarse en [NA01].

9.2.1. Una breve panorámica sobre pH

Como ya hemos comentado anteriormente, pH es un lenguaje paralelo cuyo núcleo funcional está formado por Haskell, un lenguaje puramente funcional, polimórfico en sus tipos y perezoso. Sin embargo, pH no se evalua siguiendo totalmente las reglas de la evaluación perezosa, es decir, no se evalúa a *whnf* tomando como *redex* aquél más externo y más a la izquierda. El motivo es la explotación que se hace del paralelismo.

La extensión que realiza pH con respecto a Haskell no sólo introduce paralelismo en la evaluación de expresiones, sino que incorpora elementos característicos de los lenguajes imperativos, pues sus creadores consideran ventajosos los efectos laterales por tres razones:

- Pueden incrementar la modularidad de algunos programas;

- muchos programas pueden explotar de manera efectiva el no-determinismo para eliminar restricciones;
- a menudo redundan en una mejor ejecución, ya que pueden reducir la complejidad algorítmica de algunos programas.

A diferencia de Jauja (Eden), pH no dispone de un concepto explícito de proceso ni de canales para realizar comunicaciones entre las distintas partes que se evalúan en paralelo. Sin embargo, la inclusión de celdas actualizables se realiza con la atención puesta en la sincronización implícita. Una celda es actualizable porque el valor que en ella se contiene puede no estar disponible en el momento de su creación y, en un tipo concreto de celdas, dicho valor puede ser cambiado a lo largo de la evaluación..

El núcleo sintáctico de pH, que para abreviar ya ofrecemos normalizado, aparece en la Figura 9.5.

E	$::=$	x	variable
	$\mid$	$\backslash x.E$	λ-abstracción
	$\mid$	x_1 `$` x_2	aplicación perezosa
	$\mid$	x_1 `$!` x_2	aplicación estricta
	$\mid$	`let` $\{x_i = E_i\}_{i=1}^{n}$ `in` x	declaración local
	$\mid$	`iCell` $x \mid$ `mCell` x	creación de celdas
	$\mid$	`Fetch` $x \mid$ `Store` (x_1, x_2)	operaciones sobre celdas

Figura 9.5: pH-CORE restringido

Las construcciones sintácticas básicas son las mismas que contenía Jauja, no en vano el núcleo funcional de ambos lenguajes es el mismo. Por lo tanto, la evaluación de variables, abstracciones y aplicaciones funcionales perezosas será idéntica. La nota de color la ponen la evaluación de la declaración local de variables, el tratamiento de las celdas actualizables y la introducción de otro tipo de aplicación, la estricta, que requiere no sólo que la abstracción se encuentre evaluada, sino que también el argumento se haya evaluado antes de proceder con la aplicación

Las celdas pueden ser de dos tipos, a saber, I-celdas y M-celdas. En la Figura 9.6 se sintetizan los comportamientos de ambos tipos.

	I-celda	**M-celda**
`cell`	creación de I-celda vacía	creación de M-celda vacía
`fetch`	lectura de I-celda error si vacía	lectura de M-celda error si vacía vacía tras lectura
`store`	almacenamiento en I-celda error si llena	almacenamiento en M-celda error si llena

Figura 9.6: pH-CORE: comportamiento de celdas actualizables

Las I-celdas separan la creación de una variable de la definición de su valor, de manera que cualquier intento de consultar el valor de una I-celda es erróneo hasta que dicho valor está definido. Sin embargo, cuando una I-celda se llena con un valor, la celda nunca será vaciada y el valor nunca será modificado. Por otra parte, las M-celdas se crean del mismo modo, pero una consulta de una M-celda vacía el contenido de dicha celda, de modo que no se podrá realizar ninguna consulta sin antes haber "rellenado" la M-celda otra vez.

Estas celdas actualizables introducen efectos laterales y no-determinismo:

- Una operación `Fetch` sobre una M-celda devuelve el valor almacenado en ella y, como efecto lateral, vacía dicha celda.
- Debido a *race-conditions* puede suceder que la evaluación de una expresión devuelva valores diferentes en diferentes ocasiones. Por ejemplo, para una expresión como

  $$\texttt{let}\ \ t = \texttt{mCell}\ m\,, x = \texttt{Store}\ m\, v_1, y = \texttt{Store}\ m\, v_2, z = \texttt{Fetch}\ m\ \texttt{in}\ z$$

  tanto el valor v_1 como el valor v_2 pueden ser asignados a z.

9.2.2. Dominios semánticos

La Figura 9.7 contiene los dominios semánticos necesarios para formalizar la semántica de pH-CORE. A continuación los compararemos con los que se utilizaron para definir las semánticas de Jauja y GPH-CORE.

	Cont	=	**Store** $\to$ **SStore**	continuaciones
$\kappa \in$	**ECont**	=	**EVal** $\to$ **Cont**	continuaciones de expresión
$\sigma \in$	**Store**	=	**Loc** $\to$ (**Val** + {undefined})	*stores*
$\Sigma \in$	**SStore**	=	$\mathcal{P}_f$(**Store**)	conjuntos de *stores*
$\rho \in$	**Env**	=	**Ide** $\to$ **Loc**	entornos
$v \in$	**Val**	=	**EVal** + **Clo** + **Cell** + {not_ready}	valores
$\varepsilon \in$	**EVal**	=	**Abs** + {unit}	valores de expresión
$\alpha \in$	**Abs**	=	**Loc** $\to$ **Clo**	valores de abstracción
$\nu \in$	**Clo**	=	**ECont** $\to$ **Cont**	clausuras
	Cell	=	{I, M} $\times$ (**EVal** + {empty})	celdas actualizables
$l \in$	**Loc**			localidades

Figura 9.7: pH-CORE: dominios semánticos

De manera similar a como sucedía en GPH-CORE y opuestamente a Jauja, pH-CORE no tiene ni procesos ni canales de comunicación. Sin embargo, para modelizar la separación de la creación de celdas de la definición del valor que contendrán, necesitamos un mecanismo de doble ligadura: entornos y *stores*. Las localidades de un *store* pueden contener un valor o estar indefinidas (undefined), mientras que los entornos ligan identificadores a localidades en el correspondiente *store*.

Así pues, en el caso de pH-CORE el estado de un programa se representa mediante un *store* global ($\sigma \in$ **Store**). Como explicamos previamente, las M-celdas son introductoras de no-determinismo; en consecuencia, las continuaciones transformarán un *store* en un conjunto de *stores*.

De manera similar a como sucedía para los dos lenguajes ya estudiados, el dominio de valores incluye valores de expresión, clausuras y el valor especial not_ready. Además, se incluye una nueva clase de valores: celdas (actualizables), en las que distinguimos su tipo empleando las etiquetas I y M para I-celdas y M-celdas, respectivamente. A su vez, cada celda puede estar vacía o contener un valor de expresión.

Además de seguir estando presentes los valores de abstracción —que en este caso son funciones en (**Loc** $\rightarrow$ **Clo**)— el domino de valores de expresión contiene otro valor especial, unit, que se emplea para indicar que se ha evaluado una expresión cuyo efecto no es la producción de un valor, sino la modificación del estado mediante efectos laterales, tales como crear una celda o almacenar un valor en una celda.

9.2.3. Función de evaluación

Aparte de la continuación de expresión, la función de evaluación $\mathcal{E}$ para pH-CORE necesita un entorno que determine las localidades de las variables libres de la expresión. La definición de esta función se incluye en la Figura 9.8 y su tipo es:

$$\mathcal{E} : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{ECont} \rightarrow \mathbf{Cont}$$

Para evaluar una *variable* se utiliza el mecanismo habitual: se fuerza la evaluación del valor almacenado en la correspondiente localidad. De nuevo es la función auxiliar force (incluida en la Figura 9.9) la que desempeña esta tarea. Esta función se define de manera similar a las definidas previamente para Jauja y GPH-CORE.

En el caso de las λ-*abstracciones*, se crea el correspondiente valor de abstracción y se le aplica la continuación de expresión.

En el caso de las *aplicaciones*, tanto la perezosa como la estricta, se fuerza la evaluación de la expresión que ha de dar lugar al valor de abstracción. La diferencia entre ellas reside en que, en el caso de la aplicación perezosa, el argumento se almacena en forma de clausura en una nueva localidad y solamente será evaluado bajo demanda. Por contra, en presencia de una aplicación estricta, se fuerza la aplicación del argumento antes de proceder a la aplicación denotacional.

En lo referente a las celdas actualizables, nos encontramos con tres acciones diferentes:

Creación: la variable argumento de la expresión de creación tiene asociada una localidad. Esta localidad se llena con el valor denotacional de una celda vacía cuyo tipo se refleja en la etiqueta asignada.

$$\mathcal{E}\,[\![x]\!]\,\rho\,\kappa = \mathsf{force}\,(\rho\,x)\,\kappa$$

$$\mathcal{E}\,[\![\backslash x.E]\!]\,\rho\,\kappa = \kappa(\lambda l.\mathcal{E}\,[\![E]\!]\,(\rho \oplus \{x \mapsto l\}))$$

$$\mathcal{E}\,[\![x_1\ \$\ x_2]\!]\,\rho\,\kappa = \mathcal{E}\,[\![x_1]\!]\,\rho\,\kappa'$$
$$\texttt{donde}\ \kappa' = \lambda\varepsilon.\lambda\sigma.\texttt{case}\ \varepsilon\ \texttt{of}$$
$$\varepsilon \in \mathbf{Abs} \longrightarrow \varepsilon\,l\,\kappa\,\sigma'$$
$$\texttt{donde}\ l = \mathsf{freeloc}\ \sigma$$
$$\sigma' = \sigma \oplus \{l \mapsto \mathcal{E}\,[\![x_2]\!]\,\rho\}$$
$$\mathsf{e.o.c.} \longrightarrow \mathsf{wrong}$$
$$\texttt{endcase}$$

$$\mathcal{E}\,[\![x_1\ \$!\ x_2]\!]\,\rho\,\kappa = \lambda\sigma.\ \bigcup_{\sigma_2 \in \Sigma_2} \kappa'\,(\sigma_2\,(\rho\,x_1))\,\sigma_2$$
$$\texttt{donde}\ \Sigma_1 = \mathsf{force}\,(\rho\,x_1)\,id_\kappa\,\sigma$$
$$\Sigma_2 = \bigcup_{\sigma_1 \in \Sigma_1} \mathsf{force}\,(\rho\,x_2)\,id_\kappa\,\sigma_1$$
$$\kappa' = \lambda\varepsilon.\lambda\sigma'.\texttt{case}\ \varepsilon\ \texttt{of}$$
$$\varepsilon \in \mathbf{Abs} \longrightarrow \varepsilon\,(\rho\,x_2)\,\kappa\,\sigma'$$
$$\mathsf{e.o.c.} \longrightarrow \mathsf{wrong}$$
$$\texttt{endcase}$$

$$\mathcal{E}\,[\![\texttt{let}\ \{x_i = E_i\}_n\ \texttt{in}\ x]\!]\,\rho\,\kappa = \lambda\sigma.\mathcal{E}\,[\![x]\!]\,\rho'\,\kappa'\,\sigma'$$
$$\texttt{donde}\ \{l_1,\ldots,l_n\} = \mathsf{freeloc}\ n\ \sigma$$
$$\rho' = \rho \oplus \{x_1 \mapsto l_1,\ldots,x_n \mapsto l_n\}$$
$$\sigma' = \sigma \oplus \{l_1 \mapsto \mathcal{E}\,[\![E_1]\!]\,\rho',\ldots,l_n \mapsto \mathcal{E}\,[\![E_n]\!]\,\rho'\}$$
$$\kappa' = \lambda\varepsilon.\lambda\sigma''.\ \bigcup_{\sigma_d \in \Sigma_d} \kappa\,\varepsilon\,\sigma_d$$
$$\texttt{donde}\ \Sigma_d = \mathsf{decls}\,\{x_1,\ldots,x_n\}\,\rho'\,\sigma''$$

$$\mathcal{E}\,[\![\texttt{iCell}\ x]\!]\,\rho\,\kappa = \lambda\sigma.\kappa\,\mathrm{unit}\,(\sigma \oplus \{(\rho\,x) \mapsto \langle \mathrm{I}, \mathrm{empty}\rangle\})$$

$$\mathcal{E}\,[\![\texttt{mCell}\ x]\!]\,\rho\,\kappa = \lambda\sigma.\kappa\,\mathrm{unit}\,(\sigma \oplus \{(\rho\,x) \mapsto \langle \mathrm{M}, \mathrm{empty}\rangle\})$$

$$\mathcal{E}\,[\![\texttt{Fetch}\ x]\!]\,\rho\,\kappa = \lambda\sigma.\texttt{case}\ \sigma(\rho\,x)\ \texttt{of}$$
$$\langle \mathrm{I}, \varepsilon'\rangle \longrightarrow \kappa\,\varepsilon'\,\sigma$$
$$\langle \mathrm{M}, \varepsilon'\rangle \longrightarrow \kappa\,\varepsilon'\,(\sigma \oplus \{(\rho\,x) \mapsto \langle \mathrm{M}, \mathrm{empty}\rangle\})$$
$$\mathsf{e.o.c.} \longrightarrow \mathsf{wrong}\,\sigma$$
$$\texttt{endcase}$$

$$\mathcal{E}\,[\![\texttt{Store}\,(x_1, x_2)]\!]\,\rho\,\kappa = \mathcal{E}\,[\![x_2]\!]\,\rho\,\kappa'$$
$$\texttt{donde}\ \kappa' = \lambda\varepsilon.\lambda\sigma.\kappa\,\mathrm{unit}\,\sigma'$$
$$\sigma' = \texttt{case}\ \sigma(\rho\,x_1)\ \texttt{of}$$
$$\langle \mathrm{I}, \mathrm{empty}\rangle \longrightarrow \sigma \oplus \{(\rho\,x_1) \mapsto \langle \mathrm{I}, \varepsilon\rangle\}$$
$$\langle \mathrm{M}, \mathrm{empty}\rangle \longrightarrow \sigma \oplus \{(\rho\,x_1) \mapsto \langle \mathrm{M}, \varepsilon\rangle\}$$
$$\mathsf{e.o.c.} \longrightarrow \mathsf{wrong}\,\sigma$$
$$\texttt{endcase}$$

Figura 9.8: pH-CORE: función de evaluación

Consulta: si la celda sobre la que se realiza la consulta está vacía, se producirá un error; si la celda es del tipo M-celda, ésta se vacía tras la consulta del contenido.

Almacenamiento: se procede a la evaluación del segundo argumento de `Store` y la continuación de expresión que se aplica sobre este valor se encarga de comprobar si la celda está llena o vacía. En el primer caso se almacena el valor obtenido, mientras que en el segundo se produce un error.

Se puede observar una clara similitud entre celdas y canales: el rellenado de una celda se asemeja a la acción de comunicar un valor. Por otro lado, el valor de una celda se puede consultar solamente cuando esté llena. De manera similar, la recepción de un valor solamente tiene lugar si el valor ya ha sido enviado.

La evaluación de una *declaración local* se desarrolla en dos fases:

1. Entorno y *store* son ampliados para dar cobijo a la información proveniente de las nuevas variables locales.

2. Se crean hebras paralelas para evaluar cada una de estas variables locales. Esta tarea la realiza la función decls, incluida en la Figura 9.9.

En pH el cómputo de un programa solamente termina cuando todas las hebras creadas han completado su evaluación. Consecuentemente, en este caso no procede la distinción entre semántica mínima y máxima que realizábamos en los casos anteriores; además, la definición semántica dada se corresponde con la semántica máxima.

9.2.4. Funciones semánticas auxiliares

Las funciones auxiliares empleadas para definir la función de evaluación para pH-CORE son force y decls, ambas definidas en la Figura 9.9.

$$
\begin{array}{l}
\mathsf{force} :: \mathbf{Loc} \to \mathbf{ECont} \to \mathbf{Cont} \\
\mathsf{force}\,\alpha\,\kappa = \lambda\sigma.\mathtt{case}\ (\sigma\,\alpha)\ \mathtt{of} \\
\quad \varepsilon \in \mathbf{Abs} \longrightarrow \kappa\,\varepsilon\,\sigma \\
\quad \nu \in \mathbf{Clo} \longrightarrow \nu\,\kappa'\,\sigma' \\
\qquad \mathtt{donde}\ \kappa' = \lambda\varepsilon''.\lambda\sigma''.\kappa\,\varepsilon''\,\sigma'' \oplus \{l \mapsto \varepsilon''\} \\
\qquad\qquad\ \ \sigma' = \sigma \oplus \{l \mapsto \mathrm{not\_ready}\} \\
\quad \mathsf{e.o.c.} \longrightarrow \mathsf{wrong} \\
\mathtt{endcase}
\end{array}
\qquad
\begin{array}{l}
\mathsf{decls} :: \mathcal{P}_f(\mathbf{Ide}) \to \mathbf{Env} \to \mathbf{Cont} \\
\mathsf{decls}\,\emptyset\,\rho = \lambda\sigma.\{\sigma\} \\
\mathsf{decls}\,I\,\rho = \\
\quad = \lambda\sigma.\ \bigcup_{x \in I}\ (\ \bigcup_{\sigma_x \in \Sigma_x} \mathsf{decls}\,(I \setminus \{x\})\,\rho\,\sigma_x) \\
\qquad \mathtt{donde}\ \Sigma_x = \mathcal{E}\,[\![x]\!]\,\rho\,id_\kappa\,\sigma
\end{array}
$$

Figura 9.9: pH-CORE: funciones semánticas auxiliares

La función force es similar a las presentadas con anterioridad con la salvedad de que ahora no se fuerzan identificadores, sino localidades, y, en consecuencia, todas las asociaciones nuevas se hacen a localidades.

Gracias a la función decls se consigue introducir el no-determinismo derivado de las *race-conditions*, pues se consideran todos los órdenes posibles de evaluación de las variables locales.

Finalmente, la función id_κ también ha de particularizarse para esta nueva semántica:

$$
\begin{array}{l}
id_\kappa :: \mathbf{EVal} \to \mathbf{Cont} \\
id_\kappa = \lambda\varepsilon.\lambda\sigma.\{\sigma\}
\end{array}
$$

Con esto culminan las exposiciones de las tres semánticas denotacionales para los tres tipos de paralelismo.

9.3. Ejemplos comparativos

En el Ejemplo 8.6 de la Sección 8.1.4 vimos una expresión que contenía paralelismo especulativo. En esta sección vamos a analizar el mismo ejemplo implementado tanto en GPH-CORE como en pH-CORE.

Ejemplo 9.1 *Paralelismo especulativo en* GpH-core.

La expresión a considerar es la siguiente:

$$\texttt{let } x_1 = x_2\, x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2\, x_3, x_5 = x_1 \texttt{ 'par' } x_2 \texttt{ in } x_5$$

Intuitivamente podemos deducir que si existen recursos suficientes en el sistema la variable x_1 será evaluada, igual que sucedía en Jauja, lo que se traduce en que en la semántica mínima tendremos esta variable sin evaluar, en tanto que en el sistema final de la máxima se encontrará evaluada.

Los valores iniciales de entorno y continuación de expresión son:

$$\rho_0 = \{x_i \mapsto \text{undefined}\} \oplus \{main \mapsto \mathcal{E}\,[\![\texttt{let } x_1 = x_2\, x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2\, x_3, x_5 = x_1 \texttt{ 'par' } x_2 \texttt{ in } x_5]\!]\,\}$$

$$\kappa_0 = id_\kappa$$

Calculemos el valor denotacional:

$\mathcal{E}\,[\![main]\!]\,\kappa_0\,\rho_0 =$

force $main\,\kappa_0\,\rho_0 =$

$\mathcal{E}\,[\![\texttt{let } x_1 = x_2\, x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2\, x_3, x_5 x_1 \texttt{ 'par' } x_2 \texttt{ in } x_5]\!]\,\kappa_1\,\rho_1 =$

$\kappa_1 = \lambda\varepsilon.\lambda\rho.\kappa_0\,\varepsilon\,\rho \oplus \{main \mapsto \varepsilon\}$

$\rho_1 = \rho_0 \oplus \{main \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![x_{10}]\!]\,\kappa_1\,\rho_2 =$

$\rho_2 = \rho_1 \oplus \{x_6 \mapsto \mathcal{E}\,[\![x_7\, x_8]\!]\,, x_7 \mapsto \mathcal{E}\,[\![\backslash x.x]\!]\,, x_8 \mapsto \mathcal{E}\,[\![3]\!]\,, x_9 \mapsto \mathcal{E}\,[\![x_7\, x_8]\!]\,, x_{10} \mapsto \mathcal{E}\,[\![x_6 \texttt{ 'par' } x_7]\!]\,\}$

force $x_{10}\,\kappa_1\,\rho_2 =$

$\mathcal{E}\,[\![x_6 \texttt{ 'par' } x_7]\!]\,\kappa_2\,\rho_3 =$

$\kappa_2 = \lambda\varepsilon.\lambda\rho.\kappa_1\,\varepsilon\,(\rho \oplus \{x_{10} \mapsto \varepsilon\})$

$\rho_3 = \rho_2 \oplus \{x_{10} \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![x_7]\!]\,\kappa_3\,\rho_3 =$

$\kappa_3 = \lambda\varepsilon.\lambda\rho.\kappa_2\,\varepsilon\,\rho_{par}$

$\rho_{par} = par\, x_6\, \rho$

force $x_7\,\kappa_3\,\rho_3 =$

$\mathcal{E}\,[\![\backslash x.x]\!]\,\kappa_4\,\rho_4 =$

$\kappa_4 = \lambda\varepsilon.\lambda\rho.\kappa_3\,\varepsilon\,(\rho \oplus \{x_7 \mapsto \varepsilon\})$

$\rho_4 = \rho_3 \oplus \{x_7 \mapsto \text{not\_ready}\}$

$\kappa_4\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_4 =$

$\kappa_3\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_5 =$

$\rho_5 = \rho_3 \oplus \{x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

$\kappa_2\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_{par} =$

semántica mínima

$\kappa_2\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_5 =$

$\kappa_1\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_6 =$

$\quad \rho_6 = \rho_2 \oplus \{x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

$\kappa_0\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_7 =$

$\quad \rho_7 = \rho_0 \oplus \{x_6 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_8 \mapsto \mathcal{E}\,[\![3]\!]\,, x_9 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,,$
$\qquad\qquad main \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

$id_\kappa\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_7 =$

$\rho_0 \oplus \{x_6 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_8 \mapsto \mathcal{E}\,[\![3]\!]\,, x_9 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, main \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

semántica máxima

$\kappa_2\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_8 =$

$\quad \rho_8 =$

$\quad \mathsf{par}_{\ max}\ x_6\,\rho_5 =$

$\quad \mathcal{E}\,[\![x_6]\!]\ id_\kappa\,\rho_5 =$

$\quad \mathsf{force}\ x_6\ id_\kappa\,\rho_5 =$

$\quad \mathcal{E}\,[\![x_7\ x_8]\!]\,\kappa_5\,\rho_9 =$

$\qquad \kappa_5 = \lambda\varepsilon.\lambda\rho.id_\kappa\ \varepsilon\,(\rho \oplus \{x_6 \mapsto \varepsilon\})$

$\qquad \rho_9 = \rho_5 \oplus \{x_6 \mapsto \mathrm{not\_ready}\}$

$\quad \mathcal{E}\,[\![x_7]\!]\,\kappa_6\,\rho_9 =$

$\qquad \kappa_6 = \lambda\varepsilon.\lambda\rho.\varepsilon\ x_8\,\kappa_5\,\rho$

$\quad \mathsf{force}\ x_7\,\kappa_6\,\rho_9 =$

$\quad \kappa_6\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_9 =$

$\quad \mathcal{E}\,[\![x_8]\!]\,\kappa_5\,\rho_9 =$

$\quad \mathsf{force}\ x_8\,\kappa_5\,\rho_9 =$

$\quad \mathcal{E}\,[\![3]\!]\,\kappa_7\,\rho_{10} =$

$\qquad \kappa_7 = \lambda\varepsilon.\lambda\rho.\kappa_5\ \varepsilon\,(\rho \oplus \{x_8 \mapsto \varepsilon\})$

$\qquad \rho_{10} = \rho_9 \oplus \{x_8 \mapsto \mathrm{not\_ready}\}$

$\quad \kappa_7\,3\,\rho_{10} =$

$\quad \kappa_5\,3\,\rho_{11} =$

$\qquad \rho_{11} = \rho_9 \oplus \{x_8 \mapsto 3\}$

$\quad id_\kappa\,3\,\rho_{12} =$

$\qquad \rho_{12} = \rho_5 \oplus \{x_8 \mapsto 3, x_6 \mapsto 3\}$

$\quad \rho_{12}$

$\kappa_2\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_{12} =$

$\kappa_1\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_{13} =$

$\quad \rho_{13} = \rho_2 \oplus \{x_8 \mapsto 3, x_6 \mapsto 3, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

$\kappa_0\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_{14} =$

$\quad \rho_{14} = \rho_0 \oplus \{x_6 \mapsto 3, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_8 \mapsto 3, x_9 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, main \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

$id_\kappa\,(\lambda x.\mathcal{E}\,[\![x]\!]\,)\,\rho_{14} =$

$\rho_0 \oplus \{x_6 \mapsto 3, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_8 \mapsto 3, x_9 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, main \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$

Si comparamos la semántica mínima de este ejemplo

$$\rho_0 \oplus \{x_6 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_8 \mapsto \mathcal{E}\,[\![3]\!]\,, x_9 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, main \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$$

con la que se obtuvo en el Ejemplo 8.6

$$\begin{array}{l}\{\langle \rho_0 \oplus \{x_5 \mapsto \langle p_0, \mathcal{E}\,[\![x_6 \# x_7]\!]\,\rangle, x_6 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \langle p_0, \mathcal{E}\,[\![3]\!]\,\rangle,\\ \qquad x_8 \mapsto \langle p_0, \mathcal{E}\,[\![x_6\ x_7]\!]\,\rangle, main \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\},\\ \quad \{\langle p_0, \text{unsent}, p_1\rangle, \langle p_1, \text{unsent}, p_0\rangle\}\rangle\} :\end{array}$$

debemos fijar nuestra atención en la parte del entorno de ambas, teniendo en cuenta que cada variable x_i de GpH-core se corresponde con la x_{i-1} de Jauja, para $6 \leq i \leq 10$. Obtenemos entonces que:

1. En ambos casos el valor asociado a la variable *main* es un valor de abstracción correspondiente al valor denotacional de la expresión sintáctica $\backslash x.x$.

2. En ambos casos el paralelismo propuesto por el programador ha sido ignorado por la evaluación que ha generado el valor de la variable *main*, de lo que deducimos que se trataba de paralelismo especulativo.

3. La pereza permanece en la estrategia de evaluación definida para ambos lenguajes, de modo que el valor asociado a la variable x_9 en el entorno final de GpH-core es una clausura, como sucedía con la correspondiente variable del único sistema final de Jauja.

Pasemos ahora a comparar los resultados obtenidos para las semánticas máximas. La de GpH-core es:

$$\rho_0 \oplus \{x_6 \mapsto 3, x_7 \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, x_8 \mapsto 3, x_9 \mapsto \mathcal{E}\,[\![x_7\ x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,, main \mapsto \lambda x.\mathcal{E}\,[\![x]\!]\,\}$$

y la de Jauja:

$$\{\langle \rho_0 \oplus \{x_5 \mapsto \langle 3, \emptyset\rangle, x_6 \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \langle 3, \emptyset\rangle, x_8 \mapsto \langle p_0, \mathcal{E}\,[\![x_6\ x_7]\!]\,\rangle, i \mapsto \langle 3, \emptyset\rangle, main \mapsto \langle \lambda x.\mathcal{E}\,[\![x]\!]\,, \emptyset\rangle\},$$
$$\{\langle p_1, \langle 3, \emptyset\rangle, p_0\rangle\langle p_0, \langle 3, \emptyset\rangle, p_1\rangle\}\rangle\}.$$

Deteniéndonos solamente en los entornos observamos que:

1. Como cabía esperar para la semántica máxima, el paralelismo ha sido desarrollado en ambos casos, hecho que queda patente en que la variable x_6 de GpH-core está asociada a un valor de expresión y la correspondiente de Jauja también.

2. Aún encontrándonos en un enfoque de explotación máxima del paralelismo, la estrategia de evaluación respeta la pereza en ambos casos: tanto la variable x_9 del entorno final de GpH-core como la correspondiente en el único sistema final para Jauja están asociadas a una clausura.

3. En ambos casos la búsqueda del valor para la variable *main* ha tenido éxito, encontrándose asociada en ambos entornos al valor denotacional correspondiente a la abstracción sintáctica que representa la identidad.

En esta comparación queda patente el hecho de que ambos dialectos de Haskell conservan la pereza del núcleo funcional. Veremos que en el caso de pH-CORE no sucede lo mismo pues la explotación del paralelismo implícito conduce a una pérdida de la pereza.

□

Ejemplo 9.2 *Paralelismo "especulativo" en pH.*

En este caso la expresión de pH que vamos a considerar es:

$$\texttt{let } x_1 = x_2 \,\$\, x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2 \,\$\, x_3 \texttt{ in } x_2$$

En los Ejemplos 8.6 y 9.1 la misión principal del programa era evaluar la variable x_2, sin embargo, las semánticas máximas de ambos casos "daban permiso" para proceder con la evaluación de x_1. En pH, como solamente se tiene el enfoque de paralelismo máximo, basta con que la variable x_1 sea local a la declaración que ayuda a definir la variable x_2. Pero, debido al paralelismo máximo, se evaluarán más variables que en los casos anteriores, a saber, x_3 y x_4.

Los valores iniciales del entorno, continuación de expresión y *store* son:

$$\rho_0 = \{main \mapsto l_m\}$$

$$\kappa_0 = id_\kappa$$

$$\sigma_0 = \{l_m \mapsto \mathcal{E}\,[\![\texttt{let } x_1 = x_2 \,\$\, x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2 \,\$\, x_3 \texttt{ in } x_2]\!]\,\rho_0\}$$

Procedamos a calcular el valor denotacional de este programa en pH:

$\mathcal{E}\,[\![main]\!]\,\rho_0\,\kappa_0\,\sigma_0 =$

$\mathsf{force}\,(\rho_0\ main)\,\kappa_0\,\sigma_0 =$

$\mathcal{E}\,[\![\texttt{let } x_1 = x_2 \,\$\, x_3, x_2 = \backslash x.x, x_3 = 3, x_4 = x_2 \,\$\, x_3 \texttt{ in } x_2]\!]\,\rho_0\,\kappa_1\,\sigma_1 =$

$\kappa_1 = \lambda\varepsilon.\lambda\sigma.\kappa_0\,\varepsilon\,(\sigma \oplus \{l_m \mapsto \varepsilon\})$

$\sigma_1 = \sigma_0 \oplus \{l_m \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![x_2]\!]\,\rho_1\,\kappa_2\,\sigma_2 =$

$\rho_1 = \rho_0 \oplus \{x_1 \mapsto l_1, x_2 \mapsto l_2, x_3 \mapsto l_3, x_4 \mapsto l_4\}$

$\sigma_2 = \sigma_1 \oplus \{l_1 \mapsto \mathcal{E}\,[\![x_2 \,\$\, x_3]\!]\,\rho_1, l_2 \mapsto \mathcal{E}\,[\![\backslash x.x]\!]\,\rho_1, l_3 \mapsto \mathcal{E}\,[\![3]\!]\,\rho_1, l_4 \mapsto \mathcal{E}\,[\![x_2 \,\$\, x_3]\!]\,\rho_1\}$

$\kappa_2 = \lambda\varepsilon.\lambda\sigma. \bigcup_{\sigma_d \in \Sigma_d} \kappa_1\,\varepsilon\,\sigma_d$

$\Sigma_d = \mathsf{decls}\,\{x_1, x_2, x_3, x_4\}\,\rho_1\,\sigma$

$\mathsf{force}\,(\rho_1\ x_2)\,\kappa_2\,\sigma_2 =$

$\mathcal{E}\,[\![\backslash x.x]\!]\,\rho_1\,\kappa_3\,\sigma_3 =$

$\quad \kappa_3 = \lambda\varepsilon.\lambda\sigma.\kappa_2\,\varepsilon\,(\sigma \oplus \{l_2 \mapsto \varepsilon\})$

$\quad \sigma_3 = \sigma_2 \oplus \{l_2 \mapsto \text{not\_ready}\}$

$\kappa_3\,(\lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}))\,\sigma_3 =$

$\kappa_2\,(\lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}))\,\sigma_4 =$

$\quad \sigma_4 = \sigma_2 \oplus \{l_2 \mapsto \lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\})\}$

$\bigcup_{\sigma_d \in \Sigma_d} \kappa_1\,(\lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}))\,\sigma_d =$

En este punto es donde se introduce el no-determinismo en el cálculo de la denotación. Sin embargo, como en la expresión inicial no aparecen celdas —que son las causantes del no-determinismo— simplificaremos el cómputo obviando el cálculo de todos los posibles sistemas, que a la postre serían el mismo.

$\Sigma_d = \mathsf{decls}\,\{x_1, x_2, x_3, x_4\}\,\rho_1\,\sigma_4 =$

$\quad \bigcup_{\sigma_{x_1} \in \Sigma_{x_1}} \mathsf{decls}\,\{x_2, x_3, x_4\}\,\rho_1\,\sigma_{x_1} =$

$\quad\quad \Sigma_{x_1} = \mathcal{E}\,[\![x_1]\!]\,\rho_1\,id_\kappa\,\sigma_4 =$

$\quad\quad \mathsf{force}\,(\rho\,x_1)\,id_\kappa\,\sigma_4 =$

$\quad\quad \mathcal{E}\,[\![x_2\,\$\,x_3]\!]\,\rho_1\,\kappa_4\,\sigma_5 =$

$\quad\quad\quad \kappa_4 = \lambda\varepsilon.\lambda\sigma, id_\kappa\,\varepsilon\,(\sigma \oplus \{l_1 \mapsto \varepsilon\})$

$\quad\quad\quad \sigma_5 = \sigma_4 \oplus \{l_1 \mapsto \text{not\_ready}\}$

$\quad\quad \mathcal{E}\,[\![x_2]\!]\,\rho_1\,\kappa_5\,\sigma_5 =$

$\quad\quad$ Simplificamos la definición de κ_5 porque el valor que se obtendrá será de abstracción.

$\quad\quad\quad \kappa_5 = \lambda\varepsilon.\lambda\sigma.\varepsilon\,l_5\,\kappa_4\,\sigma_6$

$\quad\quad\quad l_5 = \mathsf{freeloc}\,\sigma$

$\quad\quad\quad \sigma_6 = \sigma \oplus \{l_5 \mapsto \mathcal{E}\,[\![x_3]\!]\,\rho_1\}$

$\quad\quad \mathsf{force}\,(\rho_1\,x_2)\,\kappa_5\,\sigma_5 =$

$\quad\quad \kappa_5\,(\lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}))\,\sigma_5 =$

$\quad\quad \mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l_5\})\,\kappa_4\,\sigma_6 =$

$\quad\quad \mathsf{force}\,l_5\,\kappa_4\,\sigma_6 =$

$\quad\quad \mathcal{E}\,[\![x_3]\!]\,\rho_1\,\kappa_6\,\sigma_7 =$

$\quad\quad\quad \kappa_6 = \lambda\varepsilon.\lambda\sigma.\kappa_4\,\varepsilon\,(\sigma \oplus \{l_5 \mapsto \varepsilon\})$

$\quad\quad\quad \sigma_7 = \sigma_6 \oplus \{l_5 \mapsto \text{not\_ready}\}$

$\quad\quad \mathsf{force}\,(\rho_1\,x_3)\,\kappa_6\,\sigma_7 =$

$\quad\quad \mathcal{E}\,[\![3]\!]\,\rho_1\,\kappa_7\,\sigma_8 =$

$\quad\quad\quad \kappa_7 = \lambda\varepsilon.\lambda\sigma.\kappa_6\,\varepsilon\,(\sigma \oplus \{l_3 \mapsto \varepsilon\})$

$\quad\quad\quad \sigma_8 = \sigma_7 \oplus \{l_3 \mapsto \text{not\_ready}\}$

$\quad\quad \kappa_7\,3\,\sigma_8 =$

$\quad\quad \kappa_6\,3\,\sigma_9 =$

$\quad\quad\quad \sigma_9 = \sigma_8 \oplus \{l_3 \mapsto 3\}$

$\kappa_4\, 3\, \sigma_{10} =$

$\sigma_{10} = \sigma_9 \oplus \{l_5 \mapsto 3\}$

$id_\kappa\, 3\, \sigma_{11} =$

$\sigma_{11} = \sigma_{10} \oplus \{l_1 \mapsto 3\}$

$\{\sigma_{11}\} =$

$\{\{l_1 \mapsto 3, l_2 \mapsto \lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}), l_3 \mapsto 3, l_4 \mapsto \mathcal{E}\,[\![x_2\,\$\,x_3]\!]\,\rho_1, l_5 \mapsto 3, l_m \mapsto \text{not\_ready}\}\}$

$\mathsf{decls}\,\{x_2, x_3, x_4\}\,\rho_1\,\sigma_{11} =$

$\bigcup_{\sigma_{x_2} \in \Sigma_{x_2}} \mathsf{decls}\,\{x_3, x_4\}\,\rho_1\,\sigma_{x_2} =$

$\Sigma_{x_2} = \mathcal{E}\,[\![x_2]\!]\,\rho_1\,id_\kappa\,\sigma_{11} =$

$\mathsf{force}\,(\rho_1\,x_2)\,id_\kappa\,\sigma_{11} =$

$id_\kappa(\lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}))\,\sigma_{11} =$

$\{\sigma_{11}\}$

$\mathsf{decls}\,\{x_3, x_4\}\rho_1\,\sigma_{11} =$

$\bigcup_{\sigma_{x_3} \in \Sigma_{x_3}} \mathsf{decls}\,\{x_4\}\,\rho_1\,\sigma_{x_3} =$

$\Sigma_{x_3} = \mathcal{E}\,[\![x_3]\!]\,\rho_1\,id_\kappa\,\sigma_{11} =$

$\mathsf{force}\,(\rho_1\,x_3)\,id_\kappa\,\sigma_{11} =$

$id_\kappa\, 3\, \sigma_{11} =$

$\{\sigma_{11}\}$

$\mathsf{decls}\,\{x_4\}\,\rho_1\,\sigma_{11} =$

$\bigcup_{\sigma_{x_4} \in \Sigma_{x_4}} \mathsf{decls}\,\emptyset\,\rho_1\,\sigma_{x_4} =$

$\Sigma_{x_4} = \mathcal{E}\,[\![x_4]\!]\,\rho_1\,id_\kappa\,\sigma_{11} =$

$\mathsf{force}\,(\rho_1\,x_4)\,id_\kappa\,\sigma_{11} =$

$\mathcal{E}\,[\![x_2\,\$\,x_3]\!]\,\rho_1\,\kappa_8\,\sigma_{12} =$

$\kappa_8 = \lambda\varepsilon.\lambda\sigma.id_\kappa\,\varepsilon\,(\sigma \oplus \{l_4 \mapsto \varepsilon\})$

$\sigma_{12} = \sigma_{11} \oplus \{l_4 \mapsto \text{not\_ready}\}$

$\mathcal{E}\,[\![x_2]\!]\,\rho_1\,\kappa_9\,\sigma_{12} =$

Simplificamos la definición de κ_9 porque el valor que se obtendrá será de abstracción.

$\kappa_9 = \lambda\varepsilon.\lambda\sigma.\varepsilon\,l_6\,\kappa_8\,\sigma_{13}$

$\sigma_{13} = \sigma \oplus \{l_6 \mapsto \mathcal{E}\,[\![x_3]\!]\,\rho_1\}$

$\mathsf{force}\,(\rho_1\,x_2)\,\kappa_9\,\sigma_{12} =$

$\kappa_9\,(\lambda l.\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}))\,\sigma_{12} =$

$\mathcal{E}\,[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l_6\})\,\kappa_8\,\sigma_{13} =$

$\mathsf{force}\,l_6\,\kappa_8\,\sigma_{13} =$

$\mathcal{E}\,[\![x_3]\!]\,\rho_1\,\kappa_{10}\,\sigma_{14} =$

$\kappa_{10} = \lambda\varepsilon.\lambda\sigma.\kappa_8\,\varepsilon\,(\sigma \oplus \{l_6 \mapsto \varepsilon\})$

$\sigma_{14} = \sigma_{13} \oplus \{l_6 \mapsto \text{not\_ready}\}$

$\mathsf{force}\,(\rho_1\,x_3)\,\kappa_{10}\,\sigma_{14} =$

$\kappa_{10}\, 3\, \sigma_{14} =$

$$\kappa_8\, 3\, \sigma_{15} =$$
$$\quad \sigma_{15} = \sigma_{14} \oplus \{l_6 \mapsto 3\}$$
$$id_\kappa\, 3\, \sigma_{16} =$$
$$\quad \sigma_{16} = \sigma_{15} \oplus \{l_4 \mapsto 3\}$$
$$\{\sigma_{16}\}$$
$$\{\{l_1 \mapsto 3, l_2 \mapsto \lambda l.\mathcal{E}[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}), l_3 \mapsto 3, l_4 \mapsto 3, l_5 \mapsto 3, l_6 \mapsto 3, l_m \mapsto \text{not\_ready}\}\} = \{\sigma_d\}$$
$$\kappa_1\,(\lambda l.\mathcal{E}[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\})\,\sigma_d =$$
$$\kappa_0\,(\lambda l.\mathcal{E}[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\})\,\sigma_{17} =$$
$$\quad \sigma_{17} = \sigma_d \oplus \{l_m \mapsto \lambda l.\mathcal{E}[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}\}$$
$$\{\sigma_{17}\} =$$
$$\{\{l_1 \mapsto 3, l_2 \mapsto \lambda l.\mathcal{E}[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\}), l_3 \mapsto 3, l_4 \mapsto 3, l_5 \mapsto 3, l_6 \mapsto 3, l_m \mapsto \lambda l.\mathcal{E}[\![x]\!]\,(\rho_1 \oplus \{x \mapsto l\})\}\}$$

Pasamos ahora a comparar el resultado obtenido para pH con los que se obtuvieron para las semánticas máximas de los otros dos lenguajes en los Ejemplos 8.6 y 9.1. La de GpH-core era:

$$\rho_0 \oplus \{x_6 \mapsto 3, x_7 \mapsto \lambda x.\mathcal{E}[\![x]\!]\,, x_8 \mapsto 3, x_9 \mapsto \mathcal{E}[\![x_7\, x_8]\!]\,, x_{10} \mapsto \lambda x.\mathcal{E}[\![x]\!]\,, main \mapsto \lambda x.\mathcal{E}[\![x]\!]\,\}$$

y la de Jauja:

$$\{\langle \rho_0 \oplus \{x_5 \mapsto \langle 3, \emptyset\rangle, x_6 \mapsto \langle \lambda x.\mathcal{E}[\![x]\!]\,, \emptyset\rangle, x_7 \mapsto \langle 3, \emptyset\rangle, x_8 \mapsto \langle p_0, \mathcal{E}[\![x_6\, x_7]\!]\,\rangle, i \mapsto \langle 3, \emptyset\rangle, main \mapsto \langle \lambda x.\mathcal{E}[\![x]\!]\,, \emptyset\rangle\},$$
$$\{\langle p_1, \langle 3, \emptyset\rangle, p_0\rangle\langle p_0, \langle 3, \emptyset\rangle, p_1\rangle\}\rangle\}.$$

Recordemos que el paralelismo pretendido en todos los ejemplos era el correspondiente a la evaluación de la aplicación $x_2\ x_3$ ($x_2\ \$\ x_3$ en el caso de pH) al tiempo que se obtenía el valor denotacional de la variable x_2 para asociarlo a *main*. En los tres casos se evalúa esta aplicación, sin embargo, la variable inicial x_4 de las expresiones de todos los ejemplos solamente se ha evaluado en el caso de pH (localidad l_4), en tanto que en el entorno final GpH la correspondiente variable renombrada, x_9, está ligada a $\mathcal{E}[\![x_7\, x_8]\!]$ y en el de Jauja x_8 está ligada a $\langle p_0, \mathcal{E}[\![x_6\, x_7]\!]\,\rangle$.

□

Hemos visto que el marco denotacional común permite realizar una comparación detallada de programas escritos en los tres lenguajes. Evidentemente, podíamos haber definido para pH una semántica operacional del estilo de las definidas en el Capítulo 6 para Jauja y en [BKT00] para GpH, y utilizando entonces las tres semánticas operacionales también podíamos haber realizado la comparación. Sin embargo, aunque este tipo de semántica no se basa en los cambios producidos en una máquina abstracta, contiene muchos detalles del funcionamiento del lenguaje. Las semánticas denotacionales definidas son más simples a la hora de razonar con ellas. Por otra parte, nuestro interés se centraba en saber qué se evaluaba, información contenida en las semánticas denotacionales, y no en cómo se evaluaba, objetivo principal de las operacionales, que no se detalla en las denotacionales. Por ejemplo, la abstracción que se ha hecho de la copia de ligaduras de Jauja en su definición denotacional ha dado lugar a un proceso más simple de construcción del estado final.

En cuanto a las expresiones del lenguaje, con las semánticas denotacionales se vislumbran claramente las diferencias y coincidencias en la evaluación de las expresiones comunes. Así, se ve que en todas las semánticas aquí presentadas, para evaluar una variable no hay más que forzarla. Cuando se evalúa una λ-abstracción en todas ellas se construye el valor denotacional correspondiente para aplicarle luego la continuación de expresión. En cuanto a la aplicación perezosa, en los tres lenguajes se procede con la evaluación de la variable que tiene que dar lugar a la abstracción, para posteriormete aplicar el valor de expresión obtenido a la variable argumento —a la localidad asociada con la variable argumento en el caso de pH—. Por último, la evaluación de la declaración local muestra un comportamiento distinto para cada lenguaje, y con las definiciones locales se ve fácilmente en qué radican las diferencias:

- El caso de GpH es el más simple, únicamente incorpora las variables asociadas a sus clausuras en el entorno.
- En Jauja asoma el iceberg del paralelismo. Ya en el caso de la semántica mínima se crea la estructura de todos los procesos presentes en la declaración, aunque no hayan producido ningún valor. Y en la máxima, el paralelismo se explota hasta el punto de forzar la creación y evaluación de todos los procesos de la declaración.
- Por último, en pH la declaración local es la fuente del paralelismo, pues se evalúan todas las variables locales.

Observando las tres definiciones denotacionales se visualizan rápidamente estos comportamientos. Si la comparación fuera a través de semánticas operacionales sería más costoso ver las diferencias; por ejemplo, vimos en la semántica operacional del Capítulo 6 cómo para Jauja se introducían las variables de la declaración en un paso local (regla LET de la Figura 6.2), pero para ver qué procesos se creaban y cuándo había que esperar a ejecutar la regla global al efecto (regla PROCESS CREATION de la Figura 6.3). Esta separación en dos reglas no se presenta en la semántica denotacional, y todo el proceso se aprecia de un golpe de vista en la definición de la declaración local.

La exposición de los modelos formales finaliza en este punto. En la siguiente parte de esta memoria presentamos las conclusiones a las que hemos llegado con este trabajo.

PARTE IV

CONCLUSIONES Y TRABAJO FUTURO

CAPÍTULO 10

Conclusiones y trabajo futuro

Todo concluye, pero nada perece.
Lucius Annaeus Seneca

Se han escrito páginas y páginas tratando sobre la semántica de los lenguajes de programación funcional paralela y/o concurrente. Llega ahora el momento de dilucidar qué relevancia puede tener este trabajo. Al iniciar un viaje todo son expectativas que, al finalizarlo, pueden verse o no cumplidas, o que incluso pueden haber sido superadas. Las nuestras ya las expusimos en el Capítulo 1, es tiempo ahora de ver hasta qué punto han sido satisfechas.

10.1. Semántica operacional

Nuestro primer objetivo fue definir una semántica operacional para establecer cómo se ejecutan los programas, aunque sin basarse en el funcionamiento de una máquina abstracta. Tras el estudio de diferentes semánticas operacionales para lenguajes funcionales paralelos y concurrentes, estimamos que la definida en [BKT00] para GpH, en la que se conjugaban pereza y paralelismo, podía ser el origen de nuestra semántica operacional para Eden.

La primera versión publicada de la semántica operacional que hemos definido en esta memoria sólo incluía las construcciones sintácticas de Jauja básico [HO00]. Allí la planificación convertía algunas ligaduras ejecutables, como las que se emplean en [BKT00],

en activas, de forma que todas ellas evolucionaban. Como se ha visto en la presente memoria, el modelo final aquí presentado no cuenta con ligaduras ejecutables y la planificación decide cuáles de las activas tienen permiso para evolucionar. Además, aquel primer modelo estaba basado en el enfoque de la Sección 6.7, en el que las ligaduras se copian sin necesidad de estar evaluadas. Aquella primera definición también incluía un modelo que cuantificaba la especulación de procesos y la de ligaduras, cuantificación que complicaba sobremanera el modelo y las reglas de cómputo y que en el modelo actualmente presentado se obtiene comparando los cómputos de la semántica máxima y los de la mínima: es evidente que todo proceso que aparece en la primera y no en la segunda ha sido especulativo, pero también todo proceso que en la segunda no ha comunicado valores y sí en la primera.

Avanzando en el camino se incorporaron el resto de construcciones sintácticas que aparecen en Jauja, siguiendo aún sin evaluar las ligaduras a copiar. Esta segunda versión, que aún incluía ligaduras ejecutables, se presentó en [HO01a]. El siguiente paso fue eliminar las ligaduras ejecutables y definir el conjunto de ligaduras activas que se debían evaluar; esta versión se expuso primeramente en [HO02c], y se publicó definitivamente en [HO02b]. Cabe señalar que en esta definición la evaluación a forma normal se realizaba siguiendo un método diferente al expuesto en este trabajo: se consideraban variables especiales que, al ser demandadas, provocaban que la expresión asociada se evaluara a forma normal. Con anterioridad se intentaron otros métodos de evaluar a forma normal los valores-lista, que finalmente se revelaron demasiado complejos, como el etiquetado mediante una función especial de las expresiones que tenían que reducirse a forma normal.

Por último, se desarrolló la versión presentada en este trabajo. Con respecto a las versiones anteriormente mencionadas hay que destacar (1) que no se consideran ligaduras ejecutables, pues seleccionando de entre las activas aquéllas a las que se permite evolucionar se obtiene el mismo resultado y se simplifican las reglas; (2) que aquí se sigue el enfoque según el cual todo lo que se copia de un proceso a otro está previamente evaluado, que es el que se desea en la definición de Eden [BLOP96b]; y (3) que la evaluación a forma normal se obtiene como consecuencia de este enfoque, por lo que desaparecen las complejidades derivadas de la introducción de un nuevo tipo de variables.

La semántica operacional aquí expuesta constituye un modelo que define formalmente la semántica de las construcciones más significativas del lenguaje funcional paralelo Eden, a saber, aplicación funcional perezosa en conjunción con la impaciencia en la creación de procesos y en la producción de valores de comunicación, existencia de un operador no determinista para mezclar *streams* de comunicación, y posibilidad de definir canales dinámicos de comunicación. La modelización de toda estas características se ha conseguido gracias a los siguientes elementos:

- Un sistema de ligaduras etiquetadas que asocian variables con expresiones y que simulan las clausuras de evaluación como ya se hacía en [BKT00]. Gracias a estas ligaduras se conservan las variables asociadas a subexpresiones de la normalización y se retrasa la evaluación de una expresión hasta que se demanda la variable a la que se encuentra asociada.

- Existencia de procesos independientes, cada uno de ellos representado por un conjunto de ligaduras. Estos conjuntos modelizan el estado del cómputo.
- Dos niveles de transiciones: local y global.
- Diferentes grados de cómputo especulativo gracias a la libertad dada para definir el conjunto de ligaduras activas con permiso para evolucionar.

Según se ha definido la semántica, se ha considerado que se disfrutaba de la abundancia de recursos, pero la semántica puede adaptarse fácilmente para modelizar un sistema con un número limitado de procesadores, bastaría con definir el conjunto de ligaduras desarrollables ($\mathcal{LD}$ en la Sección 6.2.3) imponiendo un límite sobre su cardinal.

En el paradigma de paralelismo explícito es crucial tener herramientas que midan el grado de eficacia de un programa particular, de modo que puedan verse las ventajas del paralelismo introducido, en comparación con otros posibles diseños. En este sentido se han desarrollado simuladores de ejecución como Gransim [Loi96] para GpH o Paradise [Rub99] para Eden, y se han demostrado muy útiles para perfilar distintas propuestas de programas. Sin embargo, estos simuladores son, de alguna manera, comparables a probar la corrección de un programa con una batería de pruebas con respecto a demostrar la corrección de un programa. Por ello, el principal beneficio que se ha obtenido gracias a la definición de esta semántica operacional es la posibilidad de definir medidas de paralelismo desde un punto de vista teórico. ¿Por qué son tan interesantes estas medidas? Porque permiten al programador analizar desde una perspectiva formal las consecuencias de decisiones que ha tomado sobre, por ejemplo, la creación de procesos. Esta posibilidad cobra importancia en lenguajes como Eden en los que la especulación se deja en manos del programador. Las medidas definidas permiten discernir si el paralelismo introducido ha sido provechoso o si, por el contrario, solamente ha ralentizado la ejecución del programa y se ha limitado a ocupar recursos necesarios para la obtención del valor final.

Podemos decir que la presentada es una semántica pensada para implementadores de un lenguaje con las características de Jauja, o de Eden, ya que se dan las indicaciones operacionales de cómo se evalúa cada expresión de Jauja. Prueba fehaciente de este hecho es que siguiendo las reglas de la semántica se ha implementado en Haskell un intérprete de Jauja básico que construye todo el cómputo necesario para evaluar una expresión. Con la ayuda de dicho intérprete se han desarrollado los ejemplos de la Sección 6.2. Esta implementación de Jauja básico se halla disponible en la dirección web del Grupo Funcional de la Universidad Complutense de Madrid, `http://dalila.sip.ucm.es/funcional/`, en el apartado "Research reports and other material".

10.2. Semántica denotacional

Sin embargo, la semántica operacional definida contiene demasiados detalles de evaluación, lo que no la convierten en la más indicada para una persona que desee programar

en el lenguaje. Por ello, hemos definido una semántica denotacional que refleja los resultados de la evaluación sin bajar tanto el nivel de abstracción. Algunas opiniones podrían ir encaminadas a afirmar que la semántica definida no tiene un nivel de abstracción muy alto, porque no es una semántica denotacional al uso. Y están en lo cierto. Sin embargo, no es tan extraño definir una semántica denotacional con continuaciones; ya se hacía en los años 70 [Sto77, Ten76]. La elección de un modelo denotacional de continuaciones en lugar de una semántica denotacional directa nos ha permitido expresar la pereza de Jauja y los posibles *efectos laterales* producidos como resultado de la evaluación de una expresión. Ya se empleó un modelo de continuaciones para dotar de semántica a la combinación de evaluación perezosa y efectos laterales, como la impresión de resultados, en [Jos89]. En Jauja además hemos incluido paralelismo. Por ejemplo, si con Jauja queremos evaluar una creación de proceso, el valor devuelto como resultado es el mismo que si hubiéramos realizado con las mismas expresiones una aplicación funcional. Sin embargo, la intención de esta semántica denotacional no es dar a una expresión como único valor denotacional el valor producido, sino que el propósito es mostrar el paralelismo que da sentido a la existencia de Jauja. Por ello, además de obtener el valor correspondiente a la creación, se tiene que crear un proceso, considerándose esta creación y las comunicaciones subyacentes como efectos laterales a la devolución de un valor. La consideración de estos efectos laterales la lleva implícita una continuación. En definitiva, se ha definido un modelo formal para un λ-cálculo perezoso, y que además ha resultado adecuado para describir procesos distribuidos. El primer paso en la definición de esta semántica denotacional para Jauja se dio en [HO01b], posteriormente se presentó esta semántica en [HO02a] y en [HO02d]. En todas estas versiones previas el nivel de abstracción era aún más bajo que el descrito en el presente trabajo, y las definiciones de las funciones semánticas mucho más complejas. Finalmente, se elevó el nivel de abstracción obviándose detalles como la división del entorno en procesos o la copia de variables, permitiendo que las variables compartieran sus valores, siempre dentro del enfoque en el que todo lo que se copia se copia evaluado, y se presentaron estos cambios en [HO03a].

Empleando la semántica desarrollada en el Capítulo 8, hemos podido asociar cada programa con dos denotaciones: una *mínima*, que representa una evaluación casi por completo perezosa —los procesos se crean de manera impaciente, pero sus cuerpos no son evaluados hasta que no existe una demanda de evaluación sobre ellos— y una *máxima*, donde la pereza se restringe a la aplicación funcional. En consecuencia, el conjunto de todos los posibles estados resultantes de la evaluación de un programa se sitúa entre estas denotaciones mínima y máxima.

En cuanto a los usos de esta semántica denotacional, en la Sección 8.1.5 se han mostrado ejemplos de equivalencias entre expresiones, tanto en cuanto al valor final a que daba lugar su evaluación, como desde el punto de vista de la coordinación.

Pero los usos de este modelo denotacional han ido más allá de la referencia formal para un potencial programador y la demostración de propiedades. En el trabajo también hemos empleado el formalismo denotacional de continuaciones para definir la semántica formal de GpH y de pH. Estos dos lenguajes junto con Eden son representantes de los tres enfoques fundamentales de introducción de paralelismo en un lenguaje funcional, en

el caso que nos ocupa Haskell: pH de paralelismo implícito, GpH de paralelismo semi-explícito y Jauja de paralelismo explícito. De modo que un mismo marco semántico nos ha permitido comparar también estos tres paradigmas.

Las diferencias entre los tres enfoques de paralelismo se han reflejado en los dominios semánticos, donde únicamente en el caso de Jauja —paralelismo explícito— ha sido necesaria una noción de proceso. Este paralelismo explícito ha requerido dominios especiales para representar procesos y comunicaciones.

Sin embargo, no solamente varían los dominios, sino también la definición de la función semántica. El paralelismo explícito de Jauja reside en la evaluación de #-expresiones, momento en el que se crea la estructura del nuevo proceso, i.e. los canales correspondientes. Además, también en la evaluación de la declaración local `let` las diferencias aparecen claramente: GpH no necesita ningún mecanismo especial porque las nuevas variables locales se evalúan únicamente si son demandadas, por lo que basta con incorporarlas al entorno. En el caso de pH, estas variables son el punto donde se introduce el paralelismo, y, en consecuencia, todas estas variables se evalúan simultáneamente y al tiempo que se evalúa la expresión principal; gracias a la continuación de expresión se logra que todas estas variables sean evaluadas. Finalmente, Jauja solamente trata de manera especial las variables asociadas a creaciones de proceso, y otra vez es la continuación de expresión la que lleva a cabo esta tarea.

Obviamente, GpH, sin procesos ni comunicaciones ni no-determinismo ni efectos laterales, tiene una semántica mucho más simple que los otros dos lenguajes. Sin embargo, la semántica de continuaciones nos ha ayudado a detectar el paralelismo especulativo comparando el entorno final obtenido con la semántica mínima con el de la semántica máxima. Aunque muy probablemente esto también pueda ser realizado con una semántica sin continuaciones.

Por otra parte, en la Sección 9.3 se han confrontado las denotaciones de una misma expresión, particularizada para cada uno de los lenguajes, y se ha podido analizar cómo influye el paralelismo introducido en cada uno de los tres lenguajes.

El modelo semántico denotacional definido en este trabajo permite extraer el grado de paralelismo y la cantidad de cómputo especulativo. Por ejemplo, en el caso de Jauja los nodos que conforman el grafo de un conjunto de canales corresponden al número de procesos que se han creado en el sistema. Modificando las continuaciones de expresión empleadas en la semántica de la #-expresión y de la declaración local se pueden obtener otros grados de paralelismo, entre el mínimo y el máximo, dentro de este mismo marco semántico; las modificaciones irían encaminadas a demandar la evaluación de la salida, pero no del canal de entrada al proceso en el primer caso, y a evaluar sólo algunos de los procesos de la declaración local, más que los meramente demandados, pero no todos los incluidos en la declaración. En cuanto a la especulación, basta analizar las aristas del grafo: de aquellos pares de procesos unidos por dos aristas etiquetadas con unsent —o <> en el caso de *streams*— el hijo constituye un proceso especulativo.

Con respecto a los programas implementados en pH, una localidad especulativa se

caracteriza por no haber sido necesaria para la obtención del valor final. Para detectarlo se puede adaptar el modelo dado para pH definiendo una "semántica mínima". Bastaría con cambiar la continuación de expresión dada para la declaración local de variables por otra que se limitara a introducir las variables en el entorno. La comparación del *store* devuelto por la definición máxima (de la Sección 9.2) con el que devolviera esta otra definición mínima, nos permitiría deducir que las localidades que aparecieran asociadas a undefined en el entorno de la mínima, pero no así en la máxima, habrían sido especulativas en la segunda. Y, dando un paso más, cada localidad que en el *store* máximo apareciera asociada a un valor de expresión o a una celda, mientras que en el mínimo estuviera asociada a una clausura, también habría sido especulativa.

El análisis de la especulación en GpH sigue las mismas ideas que hemos esbozado para pH; basta cambiar "localidades en el *store*" por "variables en el entorno".

No obstante, el modelo denotacional también tiene limitaciones, pues su grado de abstracción no permite, por ejemplo, observar duplicación de trabajo, ya que este hecho tiene que ver con la copia de variables de un proceso a otro.

Eso sí, la semántica definida para GpH sí que nos ha permitido demostrar equivalencias que los autores de la semántica operacional del lenguaje ya deseaban establecer en [HBTK99]. Tanto estas propiedades como las que se demostraron para Jauja entran dentro del tipo de equivalencias que habría que demostrar para definir una semántica algebraica de estos lenguajes.

El marco semántico común con las tres definiciones denotacionales, de Jauja, GpH y pH, ha sido publicado en [HO03b].

Por otra parte, la semántica denotacional definida para Jauja también nos permite, observando el sistema final, extraer cierta información que la operacional no nos ofrecía. En concreto, con el modelo de continuaciones obtenemos el conjunto de canales, que nos muestra cuál ha sido la topología de procesos creada y las relaciones que se han establecido entre estos procesos.

10.3. Trabajo futuro

La sensación cuando el llega final del verano es que lo bueno también termina. Sin embargo, luego llega el otoño, lleno de proyectos y trabajos por realizar. Alguien me dijo que sería la hora de echarse una siestecilla, pero no, es el momento de volar, de iniciar nuevas líneas de investigación, y también de continuar profundizando en los temas tratados en este trabajo.

Las aplicaciones de las semánticas definidas van mucho más allá de las expuestas en el presente trabajo. En el Capítulo 1 se comentó la existencia de herramientas que daban vida a las semánticas formales de manera que las hacían ejecutables. En esta línea usaremos el marco semántico ofrecido por la lógica de reescritura, implementado

en Maude [CDE+00], que permite representar de forma natural diferentes lenguajes y modelos de computación, y en particular semánticas operacionales estructurales, como la definida en el Capítulo 6. Sin embargo, algunas de esas representaciones no son directamente ejecutables pues, por ejemplo, pueden aparecer variables nuevas en el lado derecho de las reglas, como es el caso de nuestra semántica. Esto no es un inconveniente al nivel de modelos abstractos; sin embargo, conseguir la ejecutabilidad nos permitiría obtener de forma casi inmediata un intérprete para Jauja, cuya semántica sería la "entrada" de Maude, lo que tiene sin duda un gran interés. En consecuencia, pretendemos estudiar diferentes semánticas ejecutables del lenguaje funcional paralelo Eden (Jauja), sobre las que iríamos variando las estrategias de ejecución. Dichas estrategias variarían el número de procesadores del sistema, la política de activación de ligaduras, las reglas de planificación, etc.

Tras obtener una semántica operacional ejecutable de Jauja, el paso adelante consistirá en analizar y demostrar propiedades de dicha semántica, tales como la confluencia o la terminación. Es importante resaltar que estas propiedades no se refieren a programas concretos implementados en Jauja, como las demostradas en el Capítulo 8, sino que podría decirse que son metapropiedades aplicables a todos los programas escritos en este lenguaje y que pueden depender de la estrategia de ejecución utilizada en la semántica.

Todas estas tareas serán desarrolladas en el marco del proyecto MIDAS (Metalenguajes para el Diseño y Análisis Integrado de Sistemas Móviles y Distribuidos),[1] que acaba de comenzar este año.

Queda entonces claro que las propiedades demostradas en este trabajo son solamente una muestra de las que se pueden tratar. En el futuro inmediato también se tratará el uso de estas semánticas para demostrar la corrección de las transformaciones de programas que se han definido para Eden [PPRS00], y para probar propiedades de esqueletos definidas en este lenguaje [LOP+02].

Por otra parte, en [EP02] se demuestra la corrección de la máquina STG (Spinless Tagless G-machine), máquina virtual en la que se basa en *runtime system* del compilador de Haskell GHC. Dicha demostración se realiza comparando sus reducciones con la máquina Mark-2 de Sestoft [Ses97]. Por otro lado, en [EP03] se demuestra la corrección de la máquina STG derivándola a partir de modificaciones en la semántica de Sestoft para hacerla un poco más cercana a la máquina. Una vez derivada se crea una máquina imperativa muy cercana al lenguaje C y se dan unos esquemas de generación de código para esa máquina. Todos los pasos se hacen formalmente y de todo ello se demuestra la corrección. Dando continuidad a estos trabajos, la semántica operacional que hemos definido también se empleará para demostrar la corrección de la máquina DREAM (the DistRibuted Eden Abstract Machine) [BKL+97], versión paralela de la máquina secuencial STG sobre la que se ejecuta Eden.

En cuanto a la semántica denotacional, ésta puede ser empleada para la demostración de la corrección del análisis de no-determinismo definido en [SP03], y que allí se

[1] Proyecto TIC2003-01000 financiado por el Ministerio de Ciencia y Tecnología.

demostró con respecto a una semántica denotacional de Eden que no incluía la creación de procesos, sólo se centraba en la definición del no-determinismo.

Por último, trabajaremos sobre la relación formal entre la semántica operacional (con copia evaluada) y la denotacional, restringidas en un primer momento a Jauja básico. Dicha relación se definirá sobre la topología de procesos que se crea en ambas semánticas. En la operacional, dicha estructura no se puede construir a partir del sistema final, sino que habrá que extraerla de la secuencia completa de cómputo. En la denotacional está contenida en el conjunto de canales de cada estado. Se demostrará que partiendo de un mismo sistema operacional y denotacional —convenientemente adaptados—, considerando todos los posibles cómputos operacionales y extrayendo de cada uno la topología de procesos, se tiene que este conjunto de estructuras es equivalente que el resultante de tomar de la semántica denotacional todos los conjuntos de canales del conjunto de estados finales.

Nuestro caminar por el sendero de las semánticas formales concluye aquí. Sin embargo, la senda continúa hasta confundirse con el horizonte, y ya estamos deseando pasear de nuevo por ella para descubrir qué se oculta en lontananza.

PARTE V

APÉNDICES

APÉNDICE A

Demostraciones

En este apéndice se incluyen las demostraciones del Capítulo 6.

A.1. Demostración de la Proposición 6.3

Lema A.1 *Dado un sistema S finito, entonces el número de* `#`*-expresiones en nivel superior en S es también finito.*

Demostración L.A.1
Trivial, pues si el número total de ligaduras es finito y las de creaciones de proceso son un subconjunto del total, éstas tienen que ser un número finito.

c.q.d.

Lema A.2 *Dado un sistema S con una creación de proceso factible, el número de* `#`*-expresiones tras aplicar $\xrightarrow{pc}$ decrece de manera estricta.*

Demostración L.A.2
Si se ha realizado una creación de proceso factible, $\theta \overset{\alpha}{\mapsto} x\#y$, se han introducido nuevas ligaduras en el *heap* del proceso padre y se crea un nuevo proceso. Analicemos lo que sucede en cada proceso involucrado:

- En el padre θ queda ligada al canal de salida del hijo, y se introduce el canal de entrada al hijo ligada a y, con lo que en el padre no se introduce ninguna creación de proceso de nivel superior.

- En el hijo se lleva a cabo la copia de variables del padre, pero todas están ligadas a valores *whnf*, por lo que ninguna de estas variables del *heap* inicial está ligada a una #-expresión. Por otra parte, se introducen otras dos ligaduras: una que liga el canal de salida a una aplicación funcional, que, por tanto, no es #-expresión, y la otra liga una variable al canal de entrada, por lo que tampoco se introduce una nueva creación.

De este modo, no se introducen nuevas #-expresiones en nivel superior en el sistema por la aplicación de $\xrightarrow{pc}$. En conclusión, n decrece de manera estricta tras el paso único de $\overset{pc}{\Longrightarrow}$.

Por tanto, tras aplicar $\xrightarrow{pc}$ a S el número de #-expresiones en S ha decrecido de manera estricta.

c.q.d.

Proposición 6.3
Sea S un sistema finito. El proceso de aplicación reiterada de $\xrightarrow{pc}$ a partir de S converge finitamente, por lo que la imagen de $\overset{pc}{\Longrightarrow}$ para S está bien definida.

Demostración P.6.3

Condición suficiente para que el número de aplicaciones de $\xrightarrow{pc}$ sea finito es que el número de #-expresiones en S sea finito y que éste decrezca tras la aplicación de $\xrightarrow{pc}$.

S es finito, y por el Lema A.1 su número de #-expresiones en nivel superior también es finito. Por el Lema A.2, tras aplicar de manera efectiva $\xrightarrow{pc}$, dicho número de #-expresiones decrece estrictamente.

Así, si el número de pasos simples en $\overset{pc}{\Longrightarrow}$ es 0 entonces ya es finito.

En el caso de que sea distinto de 0, procedamos por reducción al absurdo. Supongamos que el número de pasos en $\overset{pc}{\Longrightarrow}$ no es finito, es decir, el número de aplicaciones de $\xrightarrow{pc}$ no es finito. Por el Lema A.2, tras la aplicación de $\xrightarrow{pc}$ el número de #-expresiones decrece de manera estricta, y por tanto el número de #-expresiones con la aplicación de $\overset{pc}{\Longrightarrow}$ decrecería de manera infinita. Ya antes hemos deducido que el número de #-expresiones en S era finito, por lo que no puede decrecer de manera infinita. Llegado este punto de contradicción, nuestra hipótesis sobre el número de pasos simples incluidos en $\overset{pc}{\Longrightarrow}$ no puede ser cierta. Con lo que la imagen de S por $\overset{pc}{\Longrightarrow}$ está bien definida.

c.q.d.

A.2. Demostración de la Proposición 6.5

Lema A.3 *Dado un sistema finito S, entonces el número de ligaduras-esperando-canal en S es también finito.*

Demostración L.A.3
Evidente.

c.q.d.

Lema A.4 *Dado un sistema S con una comunicación factible, el número de ligaduras-esperando-canal tras aplicar $\stackrel{com}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.4
Si se ha realizado una comunicación factible, $ch \stackrel{\alpha}{\mapsto} W$, aplicando $\stackrel{com}{\longrightarrow}$, en el *heap* de proceso productor esta ligadura desaparece sin más modificación en este *heap*, y en el del proceso receptor se copia una parte del *heap* del productor, pero todas las ligaduras copiadas tienen como expresiones valores *whnf*, y la ligadura-esperando-canal tratada ya no es de este tipo, pues está ligada ahora a un valor *whnf*, por lo que el número de ligaduras-esperando-canal del sistema decrece en una. En conclusión, el número de ligaduras-esperando-canal decrece de manera estricta tras el paso único de $\stackrel{com}{\Longrightarrow}$.

c.q.d.

Proposición 6.5
Sea S un sistema finito. El proceso de aplicación reiterada de $\stackrel{com}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\stackrel{com}{\Longrightarrow}$ para S está bien definida.

Demostración P.6.5

Condición suficiente para que el número de aplicaciones de $\stackrel{com}{\longrightarrow}$ sea finito es que el número de ligaduras-esperando-canal en S sea finito y que éste decrezca tras la aplicación de $\stackrel{com}{\longrightarrow}$.

S es finito, y por el Lema A.3 su número de ligaduras-esperando-canal también es finito. Por el Lema A.4, tras aplicar de manera efectiva $\stackrel{com}{\longrightarrow}$, dicho número de ligaduras-canal decrece estrictamente. Razonando como en la Proposición 6.3 se deduce que la imagen para S por $\stackrel{com}{\Longrightarrow}$ está bien definida.

c.q.d.

A.3. Ausencia de interferencias en la planificación

El objetivo de esta sección es demostrar que no existe interferencia entre las reglas que componen la global $\stackrel{Unbl}{\Longrightarrow}$.

A.3.1. Ausencia de auto-interferencias

En primer lugar demostremos que no existen interferencias derivadas del orden de aplicación de los pasos que componen cada transición global.

Comencemos con $\overset{wUnbl}{\longrightarrow}$:

Proposición A.1 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 otro heap de S, posiblemente H_1, y $x_2 \overset{A}{\mapsto} W_2$ y $\theta_2 \overset{B}{\mapsto} E_B^{x_2}$ dos ligaduras de H_2 tal que $\theta_1 \neq \theta_2$. La aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar esta misma regla sobre θ_2.*

Demostración P.A.1
La única modificación sobre S debida a la aplicación de $\overset{wUnbl}{\longrightarrow}$ es el cambio de B a A de la ligadura $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$. En consecuencia, ni la ligadura de x_2 ni la de θ_2 son modificadas, con lo que θ_2 puede ser desbloqueada posteriormente.

c.q.d.

Estudiemos ahora la no interferencia de $\overset{deact}{\longrightarrow}$:

Proposición A.2 *Sea S un sistema, H_1 un heap de S y $\theta_1 \overset{A}{\mapsto} W_1$ una ligadura de H_1. Sea H_2 otro heap de S, posiblemente H_1, y $\theta_2 \overset{A}{\mapsto} W_2$ una ligadura de H_2 tal que $\theta_1 \neq \theta_2$. La aplicación de la regla $\overset{deact}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar esta misma regla sobre θ_2.*

Demostración P.A.2
La única modificación sobre S debida a la aplicación de $\overset{deact}{\longrightarrow}$ es el cambio de A a I de la ligadura $\theta_1 \overset{A}{\mapsto} W_1$. En consecuencia, la ligadura de θ_2 no es modificada, con lo que θ_2 puede ser desactivada posteriormente.

c.q.d.

Veamos la no interferencia de $\overset{bpc}{\longrightarrow}$:

Proposición A.3 *Sea S un sistema, H_1 un heap de S y $\theta_1 \overset{IA}{\mapsto} y_1\#z_1$ una ligadura de H_1. Sea H_2 otro heap de S, posiblemente H_1 ,y $\theta_2 \overset{IA}{\mapsto} y_2\#z_2$ una ligadura de H_2 tal que $\theta_1 \neq \theta_2$. La aplicación de la regla $\overset{bpc}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar esta misma regla sobre θ_2.*

Demostración P.A.3
La única modificación sobre S debida a la aplicación de $\overset{bpc}{\longrightarrow}$ es el cambio de A o I a I o A, respectivamente, de la ligadura $\theta_1 \overset{IA}{\mapsto} y_1\#z_1$. En consecuencia, la ligadura de θ_2 no es modificada, con lo que la creación ligada a θ_2 puede ser bloqueada posteriormente.

c.q.d.

Analicemos la imposibilidad de interferencia de $\overset{pcd}{\longrightarrow}$:

Proposición A.4 *Sea S un sistema, H_1 un heap de S, $\theta_1 \overset{B}{\mapsto} y_1\#z_1$ una ligadura de H_1 y $x_1 \overset{I}{\mapsto} E_1 \in \mathsf{nf}(y_1, H_1)$. Sea H_2 otro heap de S, posiblemente H_1, $\theta_2 \overset{B}{\mapsto} y_2\#z_2$ una ligadura de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(y_2, H_2)$, con $x_1 \neq x_2$. La aplicación de la regla $\overset{pcd}{\longrightarrow}$ sobre x_1 no impide que posteriormente se pueda aplicar esta misma regla sobre x_2.*

Demostración P.A.4
La única modificación sobre S debida a la aplicación de $\overset{pcd}{\longrightarrow}$ es el cambio de I a A de la ligadura $x_1 \overset{I}{\mapsto} E_1$. Además, este cambio afecta a lo sumo al conjunto $\mathsf{nf}(y_2, H_2)$ en el estado de esta ligadura. En consecuencia, ni la ligadura de θ_2 ni la de x_2 son modificadas, con lo que la activación de x_2 puede ser llevada a cabo posteriormente.

c.q.d.

En la Proposición A.4, si x_1 y x_2 son la misma variable, el cambio de estado provocado por θ_1 hace que el cambio que provocaría θ_2 ya esté realizado. Por lo que la única interferencia posible no inhabilita este cambio, sino que lo realiza "antes".

Por último, veamos la imposibilidad de interferencia de $\overset{vComd}{\longrightarrow}$:

Proposición A.5 *Sea S un sistema, H_1 un heap de S, $ch_1 \overset{I}{\mapsto} W_1$ una ligadura de H_1 y $x_1 \overset{I}{\mapsto} E_1 \in \mathsf{nf}(W_1, H_1)$. Sea H_2 otro heap de S, posiblemente H_1, $ch_2 \overset{I}{\mapsto} W_2$ una ligadura de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$, con $x_1 \neq x_2$. La aplicación de la regla $\overset{vComd}{\longrightarrow}$ sobre x_1 no impide que posteriormente se pueda aplicar esta misma regla sobre x_2.*

Demostración P.A.5
La única modificación sobre S debida a la aplicación de $\overset{vComd}{\longrightarrow}$ es el cambio de I a A de la ligadura $x_1 \overset{I}{\mapsto} W_1$. Además, este cambio afecta a lo sumo al conjunto $\mathsf{nf}(W_2, H_2)$ en el estado de esta ligadura. En consecuencia, ni la ligadura de ch_2 ni la de x_2 son modificadas, con lo que la activación de x_2 puede ser realizada posteriormente.

c.q.d.

En este punto es válida la observación realizada a la Proposición A.4 para el caso en el que x_1 y x_2 sean las misma variable.

Con esto damos por finalizado el análisis de las auto-interferencias.

A.3.2. Ausencia de interferencias entre distintas reglas

Además de demostrar la no interferencia en la aplicación de una regla es necesario probar que, por ejemplo, la aplicación de $\overset{wUnbl}{\longrightarrow}$ no interfiere en la aplicación de cada una de las restantes reglas de *Unbl*.

Proposición A.6 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 otro heap de S, posiblemente H_1, y $\theta_2 \overset{A}{\mapsto} W_2$ una ligadura*

de H_2. La aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{deact}{\longrightarrow}$ sobre θ_2.

Demostración P.A.6
La única modificación sobre S debida a la aplicación de $\overset{wUnbl}{\longrightarrow}$ es el cambio de B a A de la ligadura $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$. En consecuencia, la ligadura de θ_2 no es modificada, con lo que θ_2 puede ser desactivada posteriormente.

c.q.d.

Sin embargo, la aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ no aumenta las ligaduras a desactivar.

Veamos que la regla de desbloqueo no interfiere con la de bloqueo de creaciones de proceso.

Proposición A.7 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 otro heap de S, posiblemente H_1, y $\theta_2 \overset{IA}{\mapsto} y_2\#z_2$ una ligadura de H_2. La aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{bpc}{\longrightarrow}$ sobre θ_2.*

Demostración P.A.7
La única modificación sobre S debida a la aplicación de $\overset{wUnbl}{\longrightarrow}$ es el cambio de B a A de la ligadura $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$. En consecuencia, la ligadura de θ_2 no es modificada, con lo que la creación ligada a θ_2 puede ser bloqueada posteriormente.

c.q.d.

De nuevo, la regla $\overset{wUnbl}{\longrightarrow}$ no aumenta el número de posible bloqueos de creaciones. Esto es debido a la naturaleza de la expresión $E_B^{x_1}$.

Seguidamente analizamos la ausencia de interferencias provocadas por la regla de desbloqueo sobre la regla $\overset{pcd}{\longrightarrow}$.

Proposición A.8 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 otro heap de S, posiblemente H_1, $\theta_2 \overset{B}{\mapsto} y_2\#z_2$ una ligadura de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(y_2, H_2)$. La aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{pcd}{\longrightarrow}$ sobre x_2.*

Demostración P.A.8
La única modificación sobre S debida a la aplicación de $\overset{wUnbl}{\longrightarrow}$ es el cambio de B a A de la ligadura $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$. En consecuencia, la ligadura de x_2 no es modificada, con lo que demanda sobre x_2 sigue siendo posible.

c.q.d.

Nuevamente, la aplicación de la regla de desbloqueo no aumenta el número de demandas debidas a creaciones de proceso bloqueadas.

La siguiente ausencia de interferencias relaciona $\overset{wUnbl}{\longrightarrow}$ con $\overset{vComd}{\longrightarrow}$.

Proposición A.9 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 otro heap de S, posiblemente H_1, $ch_2 \overset{I}{\mapsto} W_2$ una ligadura de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$. La aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{vComd}{\longrightarrow}$ sobre x_2.*

Demostración P.A.9
La única modificación sobre S debida a la aplicación de $\overset{wUnbl}{\longrightarrow}$ es el cambio de B a A de la ligadura $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$. En consecuencia, la ligadura de x_2 no es modificada, con lo que demanda sobre x_2 sigue siendo posible.

c.q.d.

Al igual que en el resto de los casos, la aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ no habilita aplicaciones de la regla $\overset{vComd}{\longrightarrow}$ que no fueran posibles antes.

Una vez analizadas todas las interferencias (im)posibles de la regla $\overset{wUnbl}{\longrightarrow}$, pasemos a analizar las que pueda tener la regla $\overset{deact}{\longrightarrow}$ sobre el resto, comenzando por $\overset{bpc}{\longrightarrow}$.

Proposición A.10 *Sea S un sistema, H_1 un heap de S y $\theta_1 \overset{A}{\mapsto} W_1$ una ligadura de H_1. Sea H_2 otro heap de S, posiblemente H_1, y $\theta_2 \overset{IA}{\mapsto} y_2\#z_2$ una ligadura de H_2. La aplicación de la regla $\overset{deact}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{bpc}{\longrightarrow}$ sobre θ_2.*

Demostración P.A.10
La única modificación sobre S debida a la aplicación de $\overset{deact}{\longrightarrow}$ es el cambio de A a I de la ligadura $\theta_1 \overset{A}{\mapsto} W_1$. Por la naturaleza de W_1 la ligadura de θ_2 no es modificada, con lo que la creación ligada a θ_2 puede ser bloqueada posteriormente.

c.q.d.

La desactivación de ligaduras no actúa sobre ligaduras asociadas a #-expresiones, por lo que no habilita nuevos bloqueos de creaciones.

Tampoco se produce interferencia entre $\overset{deact}{\longrightarrow}$ y $\overset{pcd}{\longrightarrow}$:

Proposición A.11 *Sea S un sistema, H_1 un heap de S y $\theta_1 \overset{A}{\mapsto} W_1$ una ligadura de H_1. Sea H_2 otro heap de S, posiblemente H_1, $\theta_2 \overset{B}{\mapsto} y_2\#z_2$ una ligadura de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(y_2, H_2)$. La aplicación de la regla $\overset{deact}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{pcd}{\longrightarrow}$ sobre x_2.*

Demostración P.A.11
La única modificación sobre S debida a la aplicación de $\overset{deact}{\longrightarrow}$ es el cambio de A a I de la ligadura $\theta_1 \overset{A}{\mapsto} W_1$. La ligadura de x_2 se encuentra inactiva, por lo que no es modificada,

con lo que la demanda sobre x_2 puede ejecutarse posteriormente.

c.q.d.

El único cambio que realiza $\stackrel{deact}{\longrightarrow}$ es la desactivación de ligaduras asociadas con valores *whnf*. Dichas ligaduras no pueden encontrarse en $\mathsf{nf}(y_2, H_2)$, por lo que la desactivación no conduce a una activación por demanda posterior, es decir, la desactivación de ligaduras no habilita nuevas demandas por creación de proceso.

Finalmente, veamos cómo afecta la aplicación de la regla $\stackrel{deact}{\longrightarrow}$ sobre la posterior aplicación de $\stackrel{vComd}{\longrightarrow}$.

Proposición A.12 *Sea S un sistema, H_1 un heap de S y $\theta_1 \stackrel{A}{\mapsto} W_1$ una ligadura de H_1. Sea H_2 otro heap de S, posiblemente H_1, $ch_2 \stackrel{I}{\mapsto} W_2$ una ligadura de H_2 y $x_2 \stackrel{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$. La aplicación de la regla $\stackrel{deact}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\stackrel{vComd}{\longrightarrow}$ sobre x_2.*

Demostración P.A.12
La única modificación sobre S debida a la aplicación de $\stackrel{deact}{\longrightarrow}$ es el cambio de A a I de la ligadura $\theta_1 \stackrel{A}{\mapsto} W_1$. La ligadura de x_2 se encuentra inactiva, por lo que no es modificada, con lo que la demanda sobre x_2 puede ejecutarse posteriormente.

c.q.d.

El argumento que demostraba que $\stackrel{deact}{\longrightarrow}$ no habilitaba nuevas demandas por creación de procesos es válido para asegurar que no se habilitan nuevas demandas por comunicación.

Pasemos ahora a estudiar las interferencias que la aplicación de la regla $\stackrel{bpc}{\longrightarrow}$ puede ejercer sobre la aplicación de las reglas $\stackrel{pcd}{\longrightarrow}$ y $\stackrel{vComd}{\longrightarrow}$.

Fundamentalmente, lo que hay que justificar es por qué se realizan en este orden y no en otro. Si una #-expresión ha llegado en nivel superior a este punto es porque no tenía resueltas sus dependencias libres. Por otra parte, el estado para ejecutar una creación de proceso es irrelevante, pues una expresión de este tipo y su situación es demandada por defecto, y en cuanto se resuelvan sus dependencias será creado el nuevo proceso independientemente de en qué estado se encuentre la #-expresión. En consecuencia, no tiene sentido la activación de una creación de proceso para ejercer demanda sobre ella. Si la regla $\stackrel{pcd}{\longrightarrow}$ se ejecuta después de la regla $\stackrel{bpc}{\longrightarrow}$ no se lleva a cabo esta activación innecesaria. Sin embargo, otro orden en la aplicación de estas reglas puede conducir a activar creaciones de proceso. Es por esto que la no interferencia no consiste en ver que posteriormente se puede aplicar la otra regla sobre la otra variable. Por otra parte, la aplicación de $\stackrel{bpc}{\longrightarrow}$ y $\stackrel{pcd}{\longrightarrow}$ en cualquier orden tiene el mismo efecto sobre la ligadura $\theta_1 \stackrel{IA}{\mapsto} y_1 \# z_1$, que al final es bloqueada.

El mismo argumento es aplicable para la justificación del orden entre $\stackrel{bpc}{\longrightarrow}$ y $\stackrel{vComd}{\longrightarrow}$.

Lo que también es cierto en este caso es que la aplicación de la regla $\xrightarrow{bpc}$ no genera nuevas activaciones por las reglas $\xrightarrow{pcd}$ y $\xrightarrow{vComd}$.

Por último, veamos las relaciones existentes entre las reglas $\xrightarrow{pcd}$ y $\xrightarrow{vComd}$.

El orden entre estas dos reglas es relevante en tanto que la activación por demanda sobre una ligadura por parte de $\xrightarrow{pcd}$ puede llevar a que esta demanda no pueda ser ya ejercida por $\xrightarrow{vComd}$. Sin embargo, si pensamos en el objetivo de la aplicación de estas dos reglas, que es la demanda de las ligaduras necesarias para crear lo no creado y comunicar lo no comunicado, entonces es irrelevante quién es el que ejerce la demanda primero si al final la ligadura queda activa.

Por otra parte, la aplicación de $\xrightarrow{pcd}$ no habilita aplicaciones de $\xrightarrow{vComd}$ que no fueran posibles antes, y viceversa, la aplicación de $\xrightarrow{vComd}$ no habilita aplicaciones de $\xrightarrow{pcd}$ que antes no eran posibles, todo debido a que el único cambio que se produce en el sistema es el cambio de una ligadura inactiva a activa, es decir, no aparecen nuevas ligaduras inactivas en los conjuntos de dependencias libres.

En consecuencia, en tanto que el orden no importa y que al final todas las ligaduras que hay que demandar serán demandadas, es indistinto el orden elegido, por lo que el propuesto ($\xRightarrow{vComd} \circ \xRightarrow{pcd}$) es adecuado.

Damos por concluido el análisis de posibles interferencias en la regla $\xRightarrow{Unbl}$

A.4. Proposiciones necesarias para demostrar la Proposición 6.6

En esta sección se incluyen las proposiciones que demuestran que las reglas de la Figura 6.6 están bien definidas.

A.4.1. Demostración de que $\xRightarrow{wUnbl}$ está bien definida

Lema A.5 *Dado un sistema S finito, entonces el número de ligaduras bloqueadas en S es también finito.*

Demostración L.A.5
Trivial.

c.q.d.

Lema A.6 *Dado un sistema S con una ligadura bloqueada con posibilidad de desbloqueo, el número de ligaduras bloqueadas tras aplicar $\xrightarrow{wUnbl}$ decrece de manera estricta.*

Demostración L.A.6
Si se desbloquea una ligadura, $\theta \overset{B}{\mapsto} E^B_\rho$, tras aplicar la regla $\overset{wUnbl}{\longrightarrow}$ la ligadura bloqueada en cuestión se convierte en activa, y ninguna otra ligadura es modificada. En consecuencia, el número de ligaduras bloqueadas en S ha decrecido de manera estricta.

c.q.d.

Proposición A.13 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{wUnbl}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{wUnbl}{\Longrightarrow}$ para S está bien definida.*

Demostración P.A.13
Condición suficiente para que el número de aplicaciones de $\overset{wUnbl}{\longrightarrow}$ sea finito es que el número de ligaduras bloqueadas en S sea finito y que éste decrezca estrictamente tras la aplicación de $\overset{wUnbl}{\longrightarrow}$.

Partimos de que S es finito, y por el Lema A.5 el número de ligaduras bloqueadas en S es también finito. El Lema A.6 asegura que tras cada aplicación de $\overset{wUnbl}{\longrightarrow}$ este número de ligaduras bloqueadas decrece estrictamente. Así, siguiendo el razonamiento de casos anteriores, la imagen para S por $\overset{wUnbl}{\Longrightarrow}$ está bien definida.

c.q.d.

A.4.2. Demostración de que $\overset{deact}{\Longrightarrow}$ está bien definida

Notación A.1 Una *ligadura-whnf* es una ligadura de la forma $\theta \overset{\alpha}{\mapsto} W$. Si su estado es A la denominaremos *ligadura-whnf activa*.

□

Lema A.7 *Dado un sistema S finito, entonces el número de ligaduras-whnf en S es también finito.*

Demostración L.A.7
Trivial.

c.q.d.

Lema A.8 *Dado un sistema S con una ligadura-whnf activa (y, por consiguiente, con posibilidad de desactivación), el número de ligaduras-whnf activas tras aplicar $\overset{wUnbl}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.8
Si se desactiva una ligadura, $\theta \overset{A}{\mapsto} W$, tras aplicar la regla $\overset{deact}{\longrightarrow}$ la ligadura-*whnf* en cuestión se convierte en inactiva, y ninguna otra ligadura es modificada. En consecuencia, el número de ligaduras-*whnf* activas en S ha decrecido de manera estricta.

c.q.d.

Proposición A.14 *Sea S un sistema finito. El proceso de aplicación reiterada de $\stackrel{deact}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\stackrel{deact}{\Longrightarrow}$ para S está bien definida.*

Demostración P.A.14
Condición suficiente para que el número de aplicaciones de $\stackrel{deact}{\longrightarrow}$ sea finito es que el número de ligaduras-*whnf* activas en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\stackrel{deact}{\longrightarrow}$.

Partimos de que S es finito, y por el Lema A.7 el número de ligaduras-*whnf* activas en S es también finito. El Lema A.8 asegura que tras cada aplicación de $\stackrel{deact}{\longrightarrow}$ este número de ligaduras-*whnf* activas decrece estrictamente. Empleando el razonamiento de casos anteriores se deduce la imagen para S según $\stackrel{deact}{\Longrightarrow}$ está bien definida.

c.q.d.

A.4.3. Demostración de que $\stackrel{bpc}{\Longrightarrow}$ está bien definida

Notación A.2 Una *ligadura-#-expresión no realizada* es una ligadura de la forma $\theta \stackrel{IA}{\mapsto} x\#y$.

□

Lema A.9 *Dado un sistema S finito, entonces el número de ligaduras-#-expresión no realizadas en S es también finito.*

Demostración L.A.9
Trivial.

c.q.d.

Lema A.10 *Dado un sistema S con una ligadura-#-expresión no realizada, el número de ligaduras-#-expresión no realizadas tras aplicar $\stackrel{bpc}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.10
De manera análoga a la Demostración L. A.6.

Proposición A.15 *Sea S un sistema finito. El proceso de aplicación reiterada de $\stackrel{bpc}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\stackrel{bpc}{\Longrightarrow}$ para S está bien definida.*

Demostración P.A.15
Análoga a la Demostración P. A.13, empleando los Lemas A.9 y A.10.

A.4.4. Demostración de que $\stackrel{pcd}{\Longrightarrow}$ está bien definida

Notación A.3 Una *ligadura-#-expresión bloqueada* es una ligadura de la forma $\theta \stackrel{B}{\mapsto} x\#y$.

□

Lema A.11 *Dado un sistema S finito, entonces el número de ligaduras-#-expresión bloqueadas en S es también finito.*

Demostración L.A.11
Evidente.

c.q.d.

Notación A.4 Dado un sistema S, una *ligadura-inactiva-#-demandada en S*, $x \stackrel{I}{\mapsto} E$, es una ligadura tal que verifica que $\exists H \in S.\exists \theta \stackrel{B}{\mapsto} y\#z \in H.(x \stackrel{I}{\mapsto} E \in \mathsf{nf}(y, H))$.

□

Lema A.12 *Sea un sistema S finito, y $\theta \stackrel{B}{\mapsto} x\#y$ una ligadura-#-expresión bloqueada de H incluido en S. Entonces el número de ligaduras-inactivas-#-demandadas por $\theta \stackrel{B}{\mapsto} x\#y$ es también finito.*

Demostración L.A.12
Evidente.

c.q.d.

Lema A.13 *Dado un sistema S finito. Entonces el número de ligaduras-inactiva-#-demandadas en S es también finito.*

Demostración L.A.13
Por el Lema A.11 el número de ligaduras-#-expresión bloqueadas en S es finito. Para cada una de ellas, por el Lema A.12, sus ligaduras-inactivas-#-demandadas forman un conjunto finito. Como la unión finita de conjuntos finitos es finita se tiene que el número de ligaduras-inactiva-#-demandadas en S es también finito.

Lema A.14 *Dado un sistema S con una ligadura-#-expresión bloqueada que demande una ligadura-inactiva-#demandada. El número de ligaduras-inactivas-#-demandadas tras aplicar $\stackrel{pcd}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.14
Si se demanda una ligadura, $x \stackrel{I}{\mapsto} E$, tras aplicar la regla $\stackrel{pcd}{\longrightarrow}$, la ligadura-inactiva-#-demandada en cuestión se convierte en activa, por tanto no se incrementa el número de ligaduras-#-expresión bloqueadas, y ninguna otra ligadura es modificada. En consecuencia, el número de ligaduras-inactivas-#-demandadas en S ha decrecido de manera estricta.

c.q.d.

Proposición A.16 *Sea S un sistema finito. El proceso de aplicación reiterada de $\stackrel{pcd}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\stackrel{pcd}{\Longrightarrow}$ para S está bien definida.*

Demostración P.A.16
Condición suficiente para que el número de aplicaciones de $\stackrel{pcd}{\longrightarrow}$ sea finito es que el número de ligaduras-inactivas-#-demandadas en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\stackrel{pcd}{\longrightarrow}$.

Partimos de que S es finito, y por el Lema A.13 el número de ligaduras-inactivas-#-demandadas en S es también finito. El Lema A.14 asegura que tras cada aplicación de $\stackrel{pcd}{\longrightarrow}$ este número de ligaduras-inactivas-#-demandadas decrece estrictamente. Acudiendo al razonamiento ya habitual deducimos que la imagen para S según $\stackrel{pcd}{\Longrightarrow}$ está bien definida.

c.q.d.

A.4.5. Demostración de que $\stackrel{vComd}{\Longrightarrow}$ está bien definida

Notación A.5

- Una *ligadura-canal-inactiva* es una ligadura de la forma $ch \stackrel{I}{\mapsto} W$.
- Dado un sistema S, una *ligadura-inactiva-comunicación-demandada en S*, $x \stackrel{I}{\mapsto} E$, es una ligadura tal que verifica que $\exists H \in S. \exists ch \stackrel{I}{\mapsto} W \in H. (x \stackrel{I}{\mapsto} E \in \mathsf{nf}(W, H))$.

□

Lema A.15 *Dado un sistema S finito, entonces el número de ligaduras-comunicación-inactivas en S es también finito.*

Demostración L.A.15
Trivial.

c.q.d.

Lema A.16 *Sea un sistema S finito, y $ch \stackrel{I}{\mapsto} W$ una ligadura-canal-inactiva de H incluido en S. Entonces el número de ligaduras-inactivas-comunicación-demandada por $ch \stackrel{I}{\mapsto} W$ es también finito.*

Demostración L.A.16
Evidente.

c.q.d.

Lema A.17 *Dado un sistema S finito, el número de ligaduras-inactivas-comunicación-demandadas en S es también finito.*

Demostración L.A.17
Por el Lema A.15 el número de ligaduras-comunicación-inactivas en S es finito. Para cada una de ellas, por el Lema A.16, sus ligaduras-inactivas-comunicación-demandadas forman un conjunto finito. Como la unión finita de conjuntos finitos es finita se tiene que el número de ligaduras-inactivas-comunicación-demandadas en S es también finito.

Lema A.18 *Dado un sistema S con una ligadura-canal-inactiva que demande una ligadura-inactiva-comunicación-demandada. El número de ligaduras-inactivas-comunicación-demandadas tras aplicar $\overset{vComd}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.18
Si se demanda una ligadura, $x \overset{I}{\mapsto} E$, tras aplicar la regla $\overset{vComd}{\longrightarrow}$, la ligadura-inactiva-comunicación-demandada en cuestión se convierte en activa, y ninguna otra ligadura es modificada. Además no se genera ninguna ligadura-canal-inactiva. En consecuencia, el número de ligaduras-inactivas-comunicación-demandadas en S ha decrecido de manera estricta.

c.q.d.

Proposición A.17 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{vComd}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{vComd}{\Longrightarrow}$ para S está bien definida.*

Demostración P.A.17
Condición suficiente para que el número de aplicaciones de $\overset{vComd}{\longrightarrow}$ sea finito es que el número de ligaduras-inactivas-comunicación-demandadas en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\overset{vComd}{\longrightarrow}$.

Partimos de que S es finito, y por el Lema A.17 el número de ligaduras-inactivas-comunicación-demandadas en S es también finito. El Lema A.18 asegura que tras cada aplicación de $\overset{vComd}{\longrightarrow}$ este número de ligaduras-inactivas-comunicación-demandadas decrece estrictamente. Siguiendo la lógica de los casos anteriores, llegamos a la conclusión de que la imagen para S por $\overset{vComd}{\Longrightarrow}$ está bien definida.

c.q.d.

A.5. Demostración de la Proposición 6.9

Vamos a descomponer la demostración de la proposición referida en las demostraciones de cada caso.

A.5.1. Demostración de la Proposición 6.9 caso 1

Lema A.19 *Sea S un sistema finito. Sea H un heap de S tal que evoluciona de manera local $H \longrightarrow H'$. Entonces H' es un heap finito.*

Demostración L.A.19
Por inducción sobre la regla local (Figura 6.2) aplicada.

- Value, demand, blackhole, app-demand, β-reduction: ninguna de estas reglas introduce nuevas ligaduras en el *heap* H, por lo que el número de ligaduras de H se mantiene invariable y, por tanto, finito.
- Let: en este caso se introducen nuevas ligaduras, las correspondientes a las variables locales. Sin embargo, el número de variables locales en una declaración es finito, n. Por lo que si al cardinal finito de H le añadimos n, obteniendo el cardinal de H', éste también es finito.

De este modo cada regla local mantiene la finitud del *heap* original.

c.q.d.

Lema A.20 *Sea S un sistema finito. Sea $\langle p, H_p\rangle$ un proceso de S. Si $\langle p, H_p\rangle \stackrel{par}{\Longrightarrow}_p \langle p, H'_p\rangle$, entonces H'_p es un heap finito.*

Demostración L.A.20
El cardinal de $\mathcal{LD}(H_p)$, n_p, es finito. La evolución de cada una de estas ligaduras introduce un número finito, n_p^i, $(1 \leq i \leq n_p)$, de ligaduras nuevas en el sistema (Lema A.19). El cardinal de ligaduras nuevas introducidas en el *heap* es $\sum_{i=1}^{n_p} n_p^i$, suma de elementos finitos y, por tanto, finita. De lo que se deduce que H'_p es finito.

c.q.d.

Proposición 6.9 caso 1 *Sea S un sistema finito con $S \stackrel{par}{\Longrightarrow} S'$, entonces el sistema S' es finito.*

Demostración P.6.9 caso 1

La regla $\stackrel{par}{\Longrightarrow}$ se compone de la aplicación a cada proceso p del sistema de la regla $\stackrel{par}{\Longrightarrow}_p$. Por el Lema A.20, esta aplicación transforma cada *heap* finito de S en otro *heap* finito de S'. Por lo que S' se compone de *heaps* finitos. En consecuencia S' es un sistema finito.

c.q.d.

A.5.2. Demostración de la Proposición 6.9 caso 2

Lema A.21 *Sea S un sistema finito. Sean H_p y H_c dos heaps de S por los que se aplica la regla $\stackrel{comm}{\longrightarrow}$, transformando S en S'. Entonces S' es un sistema finito.*

Demostración L.A.21
Sean n_s el número de ligaduras de S, n_p el número de ligaduras de H_p y n_c el número de ligaduras de H_c, todos finitos porque S es un sistema finito. Tras la aplicación de $\stackrel{comm}{\longrightarrow}$ el número de ligaduras de H'_p es $n_p - 1$, el de H'_c es inferior a $n_c + n_p$, y el resto de *heaps* del sistema permanecen invariables. En consecuencia, el número de ligaduras de S crece a lo sumo en $n_p - 1$, con lo que el número de ligaduras de S' está acotado superiormente por $n_s + n_p - 1$, con lo que S' es un sistema finito.

c.q.d.

Proposición 6.9 caso 2 *Sea S un sistema finito con $S \stackrel{comm}{\Longrightarrow} S'$, entonces el sistema S' es finito.*

Demostración P.6.9 caso 2

Por el Lema A.21 cada paso simple de $\stackrel{comm}{\Longrightarrow}$ transforma un sistema finito en otro que también lo es. La Proposición 6.5 asegura que, partiendo de un sistema finito, el número de pasos simples incluidos en $\stackrel{comm}{\Longrightarrow}$ es finito. Si en cada paso el sistema crece de manera finita y hay un número finito de pasos se deduce que S' es un sistema finito.

c.q.d.

A.5.3. Demostración de la Proposición 6.9 caso 3

Lema A.22 *Sea S un sistema finito. Sea H un heap de S por el que se aplica la regla $\stackrel{pc}{\longrightarrow}$, transformando S en S'. Entonces S' es un sistema finito.*

Demostración L.A.22
Sean n_s el número de ligaduras de S y n_p el número de ligaduras de H_p, ambos finitos porque S es un sistema finito. Tras la aplicación de $\stackrel{pc}{\longrightarrow}$ el número de ligaduras de H'_p es $n_p + 1$, el de H'_c, el *heap* del nuevo proceso, es inferior a $n_p + 2$, y el resto de *heaps* del sistema permanecen invariables. En consecuencia, el número de ligaduras de S crece a lo sumo en $n_p + 3$, con lo que el número de ligaduras de S' está acotado superiormente por $n_s + n_p + 3$, deduciendo entonces que S' es un sistema finito.

c.q.d.

Proposición 6.9 caso 3 *Sea S un sistema finito con $S \stackrel{pc}{\Longrightarrow} S'$, entonces el sistema S' es finito.*

Demostración P.6.9 caso 3

Por el Lema A.22 cada paso simple de $\stackrel{pc}{\Longrightarrow}$ transforma un sistema finito en otro que también lo es. La Proposición 6.3 asegura que, partiendo de un sistema finito, el número de pasos simples incluidos en $\stackrel{pc}{\Longrightarrow}$ es finito. Si en cada paso el sistema crece de manera finita y hay un número finito de pasos se deduce que S' es un sistema finito.

c.q.d.

A.5.4. Demostración de la Proposición 6.9 caso 4

Proposición 6.9 caso 4 *Sea S un sistema finito con $S \stackrel{Unbl}{\Longrightarrow} S'$, entonces el sistema S' es finito.*

Demostración P.6.9 caso 4

Para demostrar el resultado basta con ver que cada componente de $\stackrel{Unbl}{\Longrightarrow}$ transforma un sistema finito en otro finito. Pero esto es trivial observando las reglas de la Figura 6.6, pues ninguna de ellas modifica el número de ligaduras del sistema. En consecuencia, el número de ligaduras de S finito se conserva en S', que, por tanto, es finito.

c.q.d.

A.6. Ausencia de interferencia entre comunicación simple y de *streams*

Proposición 6.10 *Sea S un sistema, H_1 un heap de S, y $ch_1 \stackrel{\alpha_1}{\mapsto} W_1 \in H_1$ una ligadura en la que W_1 es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \stackrel{\alpha_2}{\mapsto} W_2 \in H_2$ una ligadura en la que W_2 es también comunicable, verificándose $ch_1 \neq ch_2$. Entonces la aplicación de $\stackrel{vCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\stackrel{vCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.10

La realización de la comunicación de ch_1 hace que esta ligadura desaparezca y que en el proceso consumidor la ligadura bloqueada en este canal quede activa en el valor *whnf* comunicado adecuadamente renombrado.

Si $H_1 = H_2$, antes de la comunicación por ch_1 se tiene que $\mathsf{nf}(W_2, H_2) = \emptyset$; después, H_1 solamente ha cambiado en la ligadura de ch_1, con lo que $\mathsf{nf}(W_2, H_2)$ sigue siendo vacío y, por tanto, la comunicación por ch_2 es factible.

En caso de que ambos *heaps* sean distintos, la única posibilidad de que H_2 se vea afectado por la comunicación a través de ch_1 es que H_2 sea el *heap* del proceso consumidor. En otro caso no cambia y, en consecuencia, $\mathsf{nf}(W_2, H_2) = \emptyset$. Si H_2 es el consumidor, $\mathsf{nf}(W_2, H_2)$ era vacío, por lo que ahora el hecho de que una variable más esté ligada a un valor *whnf* y la incorporación de ligaduras frescas no cambian este conjunto. Así, la comunicación a través de ch_2 sigue siendo factible.

c.q.d.

Proposición 6.11 *Sea S un sistema, H_1 un heap de S, y $ch_1 \stackrel{\alpha_1}{\mapsto} [x_1^1 : x_2^1] \in H_1$ una ligadura tal que $x_1^1 \stackrel{I}{\mapsto} W_1 \in H_1 \wedge W_1$ es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \stackrel{\alpha_2}{\mapsto} [x_1^2 : x_2^2] \in H_2$ una ligadura tal que $x_1^2 \stackrel{I}{\mapsto} W_2 \in H_2 \wedge W_2$ es comunicable, verificando $ch_1 \neq ch_2$. Entonces la aplicación de $\stackrel{sCom}{\longrightarrow}$ derivada*

de ch_1 no impide la posterior aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_2.

Demostración P.6.11

Los cambios provocados por la realización de la comunicación de ch_1 son:

- En H_1, el canal ch_1 se liga a x_2^1.
- En el proceso consumidor:
  - La variable que estaba bloqueada en ch_1 se activa y se liga a una nueva lista, siendo frescas tanto la variable cabeza como la variable cola.
  - La variable cabeza se liga a W_1 convenientemente renombrado.
  - La variable cola se liga a ch_1.
  - Se copian en este *heap* las dependencias libres de W_1.

Si $H_1 = H_2$, antes de la comunicación vía ch_1 $\mathsf{nf}(W_2, H_2) = \emptyset$; después, H_1 solamente ha cambiado en la ligadura de ch_1, con lo que $\mathsf{nf}(W_2, H_2)$ sigue siendo vacío; además, no puede depender de un canal por el que se comunica, y, por tanto, la comunicación a través de ch_2 sigue siendo factible.

En caso de que ambos *heaps* sean distintos, la única posibilidad de que H_2 se vea afectado por la comunicación vía ch_1 es que H_2 sea el *heap* del proceso consumidor. En otro caso no cambia y, en consecuencia, $\mathsf{nf}(W_2, H_2) = \emptyset$. Si es el consumidor, $\mathsf{nf}(W_2, H_2)$ era vacío, por lo que ahora el hecho de que una variable más esté ligada a un valor *whnf* y la incorporación de ligaduras frescas no cambian este conjunto. Así, la comunicación a través de ch_2 continúa siendo factible.

c.q.d.

Proposición 6.12 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} W_1 \in H_1$ una ligadura en la que W_1 es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} [x_1 : x_2] \in H_2$ una ligadura tal que $x_1 \overset{I}{\mapsto} W_2 \in H_2 \wedge W_2$ es comunicable. Entonces la aplicación de $\overset{vCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.12

Remarquemos que $ch_1 \neq ch_2$, por las diferentes ligaduras de cada uno. La comunicación a través de ch_1 provoca la desaparición de este canal en H_1 y en el proceso consumidor se liga el valor *whnf* W_1 a la variable que estuviera bloqueada en este canal.

Si $H_1 = H_2$, antes de la comunicación por ch_1 $\mathsf{nf}(W_2, H_2) = \emptyset$; después, H_1 solamente ha cambiado en la ligadura de ch_1, con lo que $\mathsf{nf}(W_2, H_2)$ sigue siendo vacío; además, no puede depender de un canal por el que se comunica, y, por tanto, la comunicación por ch_2 sigue siendo factible.

En caso de que ambos *heaps* sean distintos, la única posibilidad de que H_2 se vea afectado por la comunicación a través de ch_1 es que H_2 sea el *heap* del proceso consumidor. En otro

caso no cambia y, en consecuencia, $\mathsf{nf}(W_2, H_2) = \emptyset$. Si H_2 es el consumidor, $\mathsf{nf}(W_2, H_2)$ era vacío, por lo que ahora el hecho de que una variable más esté ligada a un valor *whnf* y la incorporación de ligaduras frescas no cambian este conjunto. Así, la comunicación vía ch_2 continúa siendo factible.

c.q.d.

Proposición 6.13 *Sea S un sistema, H_1 un heap de S, y $ch_1 \overset{\alpha_1}{\mapsto} [x_1 : x_2] \in H_1$ una ligadura tal que $x_1 \overset{I}{\mapsto} W_1 \in H_1 \wedge W_1$ es comunicable. Sea H_2 un segundo heap de S, posiblemente el propio H_1, y $ch_2 \overset{\alpha_2}{\mapsto} W_2 \in H_2$ una ligadura tal que W_2 es comunicable. Entonces la aplicación de $\overset{sCom}{\longrightarrow}$ derivada de ch_1 no impide la posterior aplicación de $\overset{vCom}{\longrightarrow}$ derivada de ch_2.*

Demostración P.6.13

Recalquemos que $ch_1 \neq ch_2$, por las diferentes ligaduras de cada uno. Los cambios provocados por la realización de la comunicación de ch_1 son los indicados en la explicación de la regla HEAD-*stream* COMMUNICATION.

Si $H_1 = H_2$, antes de la comunicación vía ch_1 $\mathsf{nf}(W_2, H_2) = \emptyset$; después, H_1 solamente ha cambiado en la ligadura de ch_1, con lo que $\mathsf{nf}(W_2, H_2)$ sigue siendo vacío; además, no puede depender de un canal por el que se comunica, y, por tanto, la comunicación a través de ch_2 sigue siendo factible.

En caso de que ambos *heaps* sean distintos, la única posibilidad de que H_2 se vea afectado por la comunicación vía ch_1 es que H_2 sea el *heap* del proceso consumidor. En otro caso no cambia y, en consecuencia, $\mathsf{nf}(W_2, H_2) = \emptyset$. Si es el consumidor, $\mathsf{nf}(W_2, H_2)$ era vacío, por lo que ahora el hecho de que una variable más esté ligada a un valor *whnf* y la incorporación de ligaduras frescas no cambian este conjunto. Así, la comunicación vía ch_2 continúa siendo factible.

c.q.d.

A.7. Demostración de la Proposición 6.14

Partiendo de un sistema finito, el Lema A.3 sobre la finitud de las ligaduras-esperando-canal sigue siendo válido.

Lema A.23 *Dado un sistema S con una comunicación factible, el número de ligaduras-esperando-canal decrece de manera estricta tras aplicar $\overset{vCom}{\longrightarrow}$.*

Demostración L.A.23

Si se ha realizado una comunicación de valor simple factible, $ch \overset{\alpha}{\mapsto} W$ tal que $W \in List \Rightarrow W \equiv \mathtt{nil}$, aplicando $\overset{vCom}{\longrightarrow}$, en el *heap* del proceso productor esta ligadura desaparece sin más modificación en este *heap*, y en el del proceso receptor se copia una parte del *heap* del productor; pero todas las ligaduras copiadas tienen como expresiones valores *whnf*, y la ligadura-esperando-canal tratada ya no es de este tipo, pues está ligada

ahora a un valor *whnf*, por lo que el número de ligaduras-esperando-canal del sistema decrece en una. En conclusión, el número de ligaduras-esperando-canal decrece de manera estricta tras el paso único de $\overset{vCom}{\Longrightarrow}$.

c.q.d.

Proposición 6.14 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{vCom}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{vCom}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.14

Condición suficiente para que el número de aplicaciones de $\overset{vCom}{\longrightarrow}$ sea finito es que el número de ligaduras-esperando-canal en S sea finito y que éste decrezca tras la aplicación de $\overset{vCom}{\longrightarrow}$.

S es finito, y por el Lema A.3 su número de ligaduras-esperando-canal también es finito. Por el Lema A.23, tras aplicar de manera efectiva $\overset{vCom}{\longrightarrow}$, dicho número de ligaduras-canal decrece estrictamente. Así, siguiendo el razonamiento de casos anteriores, la imagen de S por $\overset{vCom}{\Longrightarrow}$ está bien definida.

c.q.d.

A.8. Demostración de la Proposición 6.15

Lema A.24 *Dado un sistema finito S, entonces el número de ligaduras-canal-stream en S es también finito.*

Demostración L.A.24
Trivial.

c.q.d.

Lema A.25 *Dado un sistema finito S, entonces el número de ligaduras-canal-bloqueadas-canal en S es también finito.*

Demostración L.A.25
Evidente.

c.q.d.

Lema A.26 *Dado un sistema S con una comunicación de cabeza de stream factible, la suma:*

$$n^o \text{ de ligaduras-canal-stream } + \ n^o \text{ de ligaduras-canal-bloqueadas-canal}$$

decrece de manera estricta tras aplicar $\overset{sCom}{\longrightarrow}$.

Demostración L.A.26
Si se ha realizado una comunicación factible de la cabeza de un *stream* , $ch \overset{\alpha}{\mapsto} [x_1 : x_2]$ (aplicando $\overset{sCom}{\longrightarrow}$), en el *heap* del proceso productor esta ligadura se transforma en $ch \overset{\alpha'}{\mapsto}$

x_2, por lo que ya no es ligadura-canal-*stream*. En el *heap* del proceso receptor se copia una parte del *heap* del productor, pero de todas las ligaduras copiadas ninguna tiene por expresión un canal. Además, se liga una lista a una variable. Esta variable puede ser ordinaria o de canal. El decrecimiento se da entonces porque:

- Si la ligadura de la variable receptora no era canal-bloqueada-canal, entonces el número de ligaduras-canal-*stream* ha decrecido en una y el de ligaduras-canal-bloqueadas-canal se ha mantenido igual, con lo que la suma decrece de manera estricta.
- Si dicha ligadura era canal-bloqueada-canal, entonces el número de ligaduras-canal-*stream* se ha mantenido igual pero el de ligaduras-canal-bloqueadas-canal ha decrecido en una, con lo que la suma ha decrecido de manera estricta.

En conclusión, la suma decrece de manera estricta tras el paso único de $\overset{sCom}{\Longrightarrow}$.

c.q.d.

Proposición 6.15 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{sCom}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{sCom}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.15

Condición suficiente para que el número de aplicaciones de $\overset{sCom}{\longrightarrow}$ sea finito es que la suma enunciada en el Lema A.26 en S sea finito y que éste decrezca tras la aplicación de $\overset{sCom}{\longrightarrow}$.

S es finito, y por los Lemas A.24 y A.25 la suma mencionada es finita. Por el Lema A.26, tras aplicar de manera efectiva $\overset{sCom}{\longrightarrow}$, dicha suma decrece estrictamente. Razonando como en otras proposiciones similares a ésta, la imagen para S según $\overset{sCom}{\Longrightarrow}$ está bien definida.

c.q.d.

A.9. Ausencia de interferencias con la demanda para comunicar una cabeza

Proposición 6.16 *Sea S un sistema, H_1 un heap de S, $x_1 \overset{I}{\mapsto} W_1$ una ligadura de H_1 y $x_1' \overset{I}{\mapsto} E_1 \in \mathsf{nf}(W_1, H_1)$. Sea H_2 otro heap de S, posiblemente H_1, $x_2 \overset{I}{\mapsto} W_2$ una ligadura de H_2 y $x_2' \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$, con $x_1' \neq x_2'$. La aplicación de la regla $\overset{hsComd}{\longrightarrow}$ sobre x_1' no impide que posteriormente se pueda aplicar esta misma regla sobre x_2'.*

Demostración P.6.16

La única modificación sobre S debida a la aplicación de $\overset{hsComd}{\longrightarrow}$ es el cambio de I a A de la ligadura $x_1' \overset{I}{\mapsto} W_1$. Además, este cambio afecta a lo sumo al conjunto $\mathsf{nf}(W_2, H_2)$ en el estado de esta ligadura. En consecuencia, ni la ligadura de x_2 ni la de x_2' son modificadas, con lo que la activación de x_2' puede ser realizada posteriormente.

c.q.d.

Proposición 6.17 *Sea S un sistema, H_1 un heap de S y $x_1 \overset{A}{\mapsto} W_1$ y $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$ dos ligaduras de H_1. Sea H_2 otro heap de S, posiblemente H_1, $x_2 \overset{I}{\mapsto} W_2$ una ligadura-cabeza-inactiva de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$. La aplicación de la regla $\overset{wUnbl}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{hsComd}{\longrightarrow}$ sobre x_2.*

Demostración P.6.17

La única modificación sobre S debida a la aplicación de $\overset{wUnbl}{\longrightarrow}$ es el cambio de B a A de la ligadura $\theta_1 \overset{B}{\mapsto} E_B^{x_1}$. En consecuencia, la ligadura de x_2 no es modificada, con lo que demanda sobre x_2 sigue siendo posible.

c.q.d.

Proposición 6.18 *Sea S un sistema, H_1 un heap de S y $\theta_1 \overset{A}{\mapsto} W_1$ una ligadura de H_1. Sea H_2 otro heap de S, posiblemente H_1, $x_2 \overset{I}{\mapsto} W_2$ una ligadura-cabeza-inactiva de H_2 y $x_2 \overset{I}{\mapsto} E_2 \in \mathsf{nf}(W_2, H_2)$. La aplicación de la regla $\overset{deact}{\longrightarrow}$ sobre θ_1 no impide que posteriormente se pueda aplicar la regla $\overset{hsComd}{\longrightarrow}$ sobre x_2.*

Demostración P.6.18

La única modificación sobre S debida a la aplicación de $\overset{deact}{\longrightarrow}$ es el cambio de A a I de la ligadura $\theta_1 \overset{A}{\mapsto} W_1$. La ligadura de x_2 se encuentra inactiva, por lo que no es modificada, con lo que la demanda sobre x_2 puede ejecutarse posteriormente.

c.q.d.

A.10. Demostración de la Proposición 6.19

Notación A.6

- Una *ligadura-cabeza-inactiva* es una ligadura de la forma $x_1 \overset{I}{\mapsto} W$ tal que existe en su mismo *heap* una ligadura de la forma $ch \overset{I}{\mapsto} [x_1 : x_2]$.
- Dado un sistema S, una *ligadura-inactiva-cabeza-demandada en S*, $x \overset{I}{\mapsto} E$, es una ligadura tal que

$$\exists H \in S. \exists ch \overset{I}{\mapsto} [x_1 : x_2] \in H. \exists x_1 \overset{I}{\mapsto} W. (x \overset{I}{\mapsto} E \in \mathsf{nf}(W, H)).$$

□

Lema A.27 *Dado un sistema S finito, entonces el número de ligaduras-cabeza-inactivas en S es también finito.*

Demostración L.A.27

Evidente, pues si S es finito el número de ligaduras en él contenidas es finito. Si las ligaduras-cabeza-inactivas son un subconjunto del total, entonces tienen que formar un número finito.

c.q.d.

Lema A.28 *Sea un sistema S finito, y $x \overset{I}{\mapsto} W$ una ligadura-cabeza-inactiva de H incluido en S. Entonces, el número de ligaduras-inactivas-cabeza-demandadas por $x \overset{I}{\mapsto} W$ es también finito.*

Demostración L.A.28
De nuevo es evidente, pues si S es finito y las ligaduras en cuestión están incluidas en S, entonces tienen que conformar un número finito.

c.q.d.

Lema A.29 *Dado un sistema S finito, el número de ligaduras-inactivas-cabeza-demandadas en S es también finito.*

Demostración L.A.29
Por el Lema A.27 el número de ligaduras-cabeza-inactivas de S es finito. Para cada una de ellas, por el Lema A.28 su número de ligaduras-inactivas-cabeza-demandadas en S es también finito. La suma finita de números finitos es finita, y, por tanto, el número de ligaduras-inactivas-cabeza-demandadas en S es finito.

c.q.d.

Lema A.30 *Dado un sistema S con una ligadura-cabeza-inactiva que demande una ligadura-inactiva-cabeza-demandada. El número de ligaduras-inactivas-cabeza-demandadas tras aplicar $\overset{hsComd}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.30
Si se demanda una ligadura, $x \overset{I}{\mapsto} E$, tras aplicar la regla $\overset{hsComd}{\longrightarrow}$, la ligadura-inactiva-cabeza-demandada en cuestión se convierte en activa, y ninguna otra ligadura es modificada. Además no se genera ninguna ligadura-cabeza-inactiva. En consecuencia, el número de ligaduras-inactivas-cabeza-demandadas en S ha decrecido de manera estricta.

c.q.d.

Proposición 6.19 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{hsCom}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{hsCom}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.19

Condición suficiente para que el número de aplicaciones de $\overset{hsComd}{\longrightarrow}$ sea finito es que el número de ligaduras-inactivas-cabeza-demandadas en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\overset{hsComd}{\longrightarrow}$.

Partimos de que S es finito, y por el Lema A.29 el número de ligaduras-inactivas-cabeza-demandadas en S es también finito. El Lema A.30 asegura que tras cada aplicación de $\overset{hsComd}{\longrightarrow}$ este número de ligaduras-inactivas-cabeza-demandadas decrece estrictamente. Siguiendo un razonamiento análogo al realizado para este tipo de proposición en otros casos se llega a la conclusión de que la imagen de S por $\overset{hsComd}{\Longrightarrow}$ está bien definida.

c.q.d.

A.11. Demostración de la Proposición 6.21

Notación A.7 Una *ligadura-⋈-bloqueada* es una ligadura de la forma $\theta \overset{B}{\mapsto} E^x_{\bowtie}$.

□

Lema A.31 *Dado un sistema finito S, el número de ligaduras-⋈-bloqueadas en S es finito.*

Demostración L.A.31
Como en otros casos, es evidente.

c.q.d.

Lema A.32 *Sea S un sistema con una ligadura-⋈-bloqueada que puede ser desbloqueada por la regla $\overset{mUnbl1}{\longrightarrow}$. Entonces, el número de ligaduras-⋈-bloqueadas tras aplicar $\overset{mUnbl1}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.32
Si se desbloquea una ligadura, $\theta \overset{B}{\mapsto} E^x_{\bowtie}$, tras aplicar la regla $\overset{mUnbl1}{\longrightarrow}$, entonces el estado de esta ligadura pasa de ser b a ser A. Además ninguna otra ligadura es modificada, por lo que el número de ligaduras-⋈-bloqueadas en S ha decrecido en una, es decir, el número de este tipo de ligaduras en S ha decrecido de manera estricta.

c.q.d.

Proposición 6.21 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{mUnbl1}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{mUnbl1}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.21

Condición suficiente para que el número de aplicaciones de $\overset{mUnbl1}{\Longrightarrow}$ sea finito es que el número de ligaduras-⋈-bloqueadas en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\overset{mUnbl1}{\Longrightarrow}$.

Partimos de que S es finito, y por el Lema A.31 el número de ligaduras-⋈-bloqueadas en S es también finito. El Lema A.32 asegura que tras cada aplicación de $\overset{mUnbl1}{\longrightarrow}$ este número de ligaduras-⋈-bloqueadas decrece estrictamente. Razonando de manera análoga a como se ha hecho en otras ocasiones, se demuestra que la imagen de S por $\overset{mUnbl1}{\Longrightarrow}$ está bien definida.

c.q.d.

A.12. Demostración de la Proposición 6.22

Lema A.33 *Sea S un sistema con una ligadura-⋈-bloqueada que puede ser desbloqueada por la regla $\overset{mUnbl2}{\longrightarrow}$. Entonces, el número de ligaduras-⋈-bloqueadas tras aplicar $\overset{mUnbl2}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.33
Si se desbloquea una ligadura, $\theta \overset{B}{\mapsto} E^x_{\bowtie}$, tras aplicar la regla $\overset{mUnbl2}{\longrightarrow}$, entonces el estado de esta ligadura pasa de ser B a ser A. Además ninguna otra ligadura es modificada, por lo que el número de ligaduras-$\bowtie$-bloqueadas en S ha decrecido en una, es decir, el número de este tipo de ligaduras en S ha decrecido de manera estricta.

c.q.d.

Proposición 6.22 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{mUnbl2}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{mUnbl2}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.22

Condición suficiente para que el número de aplicaciones de $\overset{mUnbl2}{\Longrightarrow}$ sea finito es que el número de ligaduras-$\bowtie$-bloqueadas en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\overset{mUnbl2}{\Longrightarrow}$.

Partimos de que S es finito, y por el Lema A.31 el número de ligaduras-$\bowtie$-bloqueadas en S es también finito. El Lema A.33 asegura que tras cada aplicación de $\overset{mUnbl2}{\longrightarrow}$ este número de ligaduras-$\bowtie$-bloqueadas decrece estrictamente. El razonamiento ya habitual demuestra que la imagen de S por $\overset{mUnbl2}{\Longrightarrow}$ está bien definida.

c.q.d.

A.13. Demostración de la Proposición 6.23

Notación A.8 Una *ligadura-esperando-conexión* es una ligadura de la forma $x \overset{B}{\mapsto} \lightning$.

□

Lema A.34 *Dado un sistema finito S, el número de ligaduras-esperando-conexión en S es finito.*

Demostración L.A.34
Trivial.

c.q.d.

Lema A.35 *Sea S un sistema con una ligadura-esperando-conexión la cual queda bloqueada en un canal tras aplicarle la regla $\overset{conn}{\longrightarrow}$. Entonces, el número de ligaduras-esperando-conexión tras aplicar $\overset{conn}{\longrightarrow}$ decrece de manera estricta.*

Demostración L.A.35
Si se una de las ligaduras mencionadas queda bloqueada en canal tras serle aplicada la regla $\overset{conn}{\longrightarrow}$, entonces esta ligadura deja de ser una ligadura-esperando-conexión. Además, del resto de ligaduras que se modifican al aplicar esta regla, ninguna de ellas queda

ligada al valor especial $\lightning$, por lo que el número de ligaduras-esperando-conexión en S ha decrecido en una, es decir, el número de este tipo de ligaduras en S ha decrecido de manera estricta.

c.q.d.

Proposición 6.23 *Sea S un sistema finito. El proceso de aplicación reiterada de $\overset{conn}{\longrightarrow}$ a partir de S converge finitamente, por lo que la imagen de $\overset{conn}{\Longrightarrow}$ para S está bien definida.*

Demostración P.6.23

Condición suficiente para que el número de aplicaciones de $\overset{conn}{\Longrightarrow}$ sea finito es que el número de ligaduras-esperando-conexión en S sea finito y que éste decrezca de manera estricta tras la aplicación de $\overset{conn}{\Longrightarrow}$.

Partimos de que S es finito, y por el Lema A.34 el número de ligaduras-esperando-conexión en S es también finito. El Lema A.35 asegura que tras cada aplicación de $\overset{conn}{\longrightarrow}$ este número de ligaduras-esperando-conexión decrece estrictamente. Razonando de manera análoga a como se ha hecho en otras ocasiones, se demuestra que la imagen de S por $\overset{conn}{\Longrightarrow}$ está bien definida.

c.q.d.

A.14. Demostración de la Proposición 6.24

Lema A.36 *Si $H^1 + H^2 : \theta \overset{A}{\mapsto} E \longrightarrow H^1 + K$ y $\theta' \overset{A}{\mapsto} E' \in H^1 \cup H^2$ con $\theta \neq \theta'$, entonces $\theta' \overset{A}{\mapsto} E' \notin K$.*

Demostración L. A.36

Si observamos todas las reglas locales de la semántica, cuando evoluciona una ligadura activa, solamente si se aplica una regla de demanda o una de bloqueo de mezcla se encuentra involucrada otra ligadura activa del *heap*. En el primer caso, las reglas son DEMAND, APP-DEMAND, Λ-DEMAND, *stream* DEMAND y CHANNEL NAME DEMAND, sin embargo, la aplicación de estas reglas únicamente modifica la ligadura activa que dirige la evolución. Y en el caso de las reglas BLOCKING MERGE sucede de la misma manera.

c.q.d.

Lema A.37 *Sea $\theta \overset{A}{\mapsto} E \in H$ una ligadura activa en la que E no es una $\bowtie$-expresión con valores en las cabezas de ambas variable. Entonces existe a lo sumo una regla local aplicable a ella.*

Demostración L. A.37

En primer lugar, la observación "a lo sumo" viene motivada por la existencia de #-expresiones —que no evolucionan por medio de ninguna regla local— así como sucede con las abstracciones y la lista vacía, que por ser valores *whnf* no tienen que evolucionar de manera local, y a lo sumo se procede a su desactivación en pasos globales. Para el resto de las expresiones existe una única regla local aplicable a cada una de ellas:

E es una variable: caben tres posibilidades, que la ligadura de la variable sea la misma que la activa a evolucionar, que sea distinta y esté inactiva en un valor *whnf* o que sea distinta y esté ligada a una expresión no en *whnf*. En el primer caso solamente se puede aplicar la regla BLACKHOLE, en el segundo la única regla local aplicable es VALUE y en el tercero la única posibilidad es aplicar la regla DEMAND.

E es una aplicación funcional: se puede estar en dos situaciones, que aún no se haya obtenido una abstracción para la primera variable o que dicha variable sí se encuentre ligada a una abstracción. En el primer caso solamente se puede aplicar la regla APP-DEMAND para pedir que se evalúe esta variable. En el segundo caso caben varias posibilidades:

1. La abstracción es una λ-abstracción, pudiéndose aplicar entonces solamente la regla β-REDUCTION.
2. La abstracción es de listas, i.e. Λ-abstracción, entonces de distinguen tres casos:
   a) La expresión ligada a la segunda variable se encuentra aún sin evaluar, con lo que la única regla que se puede aplicar es Λ-DEMAND.
   b) La expresión de la segunda variable es la lista vacía, pudiéndose aplicar solamente B-REDUCTION EMPTY LIST.
   c) La expresión asociada a la segunda variable es una lista no vacía, con lo que la única regla que se puede aplicar es B-REDUCTION NON-EMPTY LIST.

E es una declaración local de variables: la única regla local que se puede aplicar es LET.

E es una declaración de canal: la única regla local aplicable es CHANNEL CREATION.

E es una conexión dinámica: la única regla que se puede aplicar es CHANNEL NAME DEMAND.

E es una mezcla en la que a lo sumo una de las cabezas está ligada a un valor: si el valor de la cabeza no tiene dependencias libres sin resolver, entonces solamente LEFT MERGE o RIGHT MERGE, la que corresponda, es aplicable. Si ambas listas son vacías la regla MERGE TERMINATION es la única que se puede aplicar. En otro caso la mezcla no puede evolucionar, y observando las condiciones excluyentes de las reglas BLOCKING MERGE se deduce que solamente una de ellas podrá ser aplicada, a lo sumo.

E es una lista no vacía: solamente si θ es una variable de canal se podría evolucionar de manera local, y solamente podría ser por aplicación de la regla *stream* DEMAND.

En conclusión, a lo sumo se puede aplicar una regla local a la ligadura $\theta \overset{A}{\mapsto} E$.

c.q.d.

Proposición 6.24 *Consideremos dos variables distintas, θ_1 y θ_2, y un heap que puede expresarse de dos modos distintos: $H = H^{(i,1)} + H^{(i,2)} + \{\theta_i \overset{A}{\mapsto} E_i\}$, con $i \in \{1,2\}$, de modo que en cada caso tengamos $H^{(i,1)} + H^{(i,2)} : \theta_i \overset{A}{\mapsto} E_i \longrightarrow H^{(i,1)} + K^{(i,2)}$. Entonces*

1. *Si* $\theta' \overset{IB}{\mapsto} E' \in K^{(1,2)}$ *tendremos* $\theta' \notin dom(K^{(2,2)})$
2. *Si* $\theta' \overset{A}{\mapsto} E' \in K^{(1,2)}$ *tendremos* $\theta' \notin dom(K^{(2,2)})$ *o bien* $\theta' \overset{A}{\mapsto} E' \in K^{(2,2)}$

Demostración P.6.24

Distinguimos primero casos sobre la regla aplicada para $i = 1$:

- VALUE: entonces $K^{(1,2)} = \{\theta_1 \overset{A}{\mapsto} W\}$. Así, ninguna ligadura inactiva o bloqueada pertenece a este conjunto. Además, como $\theta_1 \neq \theta_2$, por el Lema A.36 la ligadura de θ_1 no puede ser modificada por la evolución de θ_2.

- DEMAND: en este caso, $K^{(1,2)} \subseteq \{\theta_1 \overset{B}{\mapsto} x, x \overset{A}{\mapsto} E'\}$. Veamos qué puede suceder con x por inducción estructural sobre E_2 (considerando que la etiqueta anterior de la ligadura de x era inactiva, se deduce además que $x \neq \theta_2$):
  - En el caso de que E_2 sea una abstracción o la lista vacía no existe una regla local aplicable.
  - Si $E_2 = y$, entonces solamente son aplicables tres reglas. Si es VALUE, entonces $K^{(2,2)} = \{\theta_2 \overset{A}{\mapsto} W\}$; y como $x \neq \theta_2$, $x \notin dom(K^{(2,2)})$. Si la regla es DEMAND, entonces $dom(K^{(2,2)}) = \{y, \theta_2\}$; la hipótesis inicial asegura que $x \neq \theta_2$, y si $x = y$ en todo caso $x \overset{A}{\mapsto} E' \in K^{(2,2)}$. Finalmente, si la regla es BLACKHOLE, entonces $K^{(2,2)} = \{\theta_2 \overset{B}{\mapsto} \theta_2\}$ y por hipótesis $x \neq \theta_2$, con lo que $x \notin dom(K^{(2,2)})$.
  - Si $E_2 = z\, y$ entonces
    - Si $z \overset{I}{\mapsto} \backslash x.E'$: la regla aplicable es β-REDUCTION, siendo entonces $K^{(2,2)} = \{\theta_2 \overset{A}{\mapsto} E'[y/x]\}$. Como $x \neq \theta_2$, entonces $x \notin dom(K^{(2,2)})$.
    - Si $z \overset{IAB}{\mapsto} = E_2^1$, y E_2^1 no es una abstracción. Entonces, la regla aplicable es APP-DEMAND, y $K^{(2,2)} \subseteq \{z \overset{A}{\mapsto} E_2^1, \theta_2 \overset{B}{\mapsto} z\, y\}$. Por hipótesis sabemos que $x \neq \theta_2$. Si $x \neq z$ entonces ya tenemos el resultado, pero si $x = z$, en ambos casos se tiene su ligadura modificada activada, con lo que también se llega al resultado deseado.
    - Si $z \overset{I}{\mapsto} = \Lambda[x_1 : x_2].E_1' \| E_2'$: se distinguen casos dependiendo de la ligadura de y.
      - Si $y \overset{IAB}{\mapsto} = E_2^2$, y E_2^2 no es una lista, entonces es aplicable es Λ-DEMAND, y $K^{(2,2)} \subseteq \{y \overset{A}{\mapsto} E_2^2, \theta_2 \overset{B}{\mapsto} z\, y\}$. Por hipótesis $x \neq \theta_2$. En el caso de que $x \neq y$ también se tiene el resultado, pero si $x = y$, la modificación de su ligadura ha sido una activación, con lo que también se tiene el resultado.
      - Si $y \overset{I}{\mapsto} = \texttt{nil}$ entonces se aplica la regla B-REDUCTION EMPTY LIST. El conjunto $K^{(2,2)}$ se reduce a la ligadura de θ_2, que, por hipótesis, es distinta de x, teniéndose así el resultado.
      - Si $y \overset{I}{\mapsto} = [y_1 : y_2]$ entonces se aplica la regla B-REDUCTION NON-EMPTY LIST. El conjunto $K^{(2,2)}$ también se reduce a la ligadura de θ_2, que, por hipótesis, es distinta de x, teniéndose así el resultado.

- Si $E_2 = \texttt{let}\ \{x^i = E^i\}_{i=1}^n\ \texttt{in}\ E'$, entonces aplicando la regla LET se tiene que $dom(K^{(2,2)}) = \{y^i\}_{i=1}^n \cup \{\theta_2\}$. $x \neq \theta_2$ y todas las variables y^i son frescas, de manera que $x \notin dom(K^{(2,2)})$.
- Si $E_2 = \texttt{new}(y_1, y_2)E'$, entonces tras aplicar la regla CHANNEL CREATION se tiene que $dom(K^{(2,2)}) = \{\theta_2, y_1', y_2'\}$. $x \neq \theta_2$ y las variables y_1', y_2' son frescas, de manera que $x \notin dom(K^{(2,2)})$.
- Si $E_2 = y_1\,!\,y_2\ \texttt{par}\ y_3$, tal que $y_1 \overset{alpha}{\mapsto} E'$ donde E' no es un valor de nombre de canal, entonces se aplica la regla CHANNEL NAME DEMMAND. Se tiene que $dom(K^{(2,2)}) \subseteq \{\theta_2, y_1\}$. Por hipótesis, $x \neq \theta_2$. En cuanto a y_1, si es distinta de x se tiene el resultado. En el caso de que $x = y_1$, entonces en ambos casos la modificación de su ligadura es una activación, con lo que se tiene el resultado.
- Si $E_2 = y_1 \bowtie y_2$, entonces varias reglas son aplicables:
  - ▷ LEFT/RIGHT MERGE: entonces $dom(K^{(2,2)}) = \{\theta_2, z\}$. Por hipótesis $x \neq \theta_2$, y como z es fresca es distinta también de x, teniéndose entonces el resultado.
  - ▷ MERGE TERMINATION o cualquiera de las reglas BLOCKING MERGE: en estos casos $dom(K^{(2,2)}) = \{\theta_2\}$, y por hipótesis $x \neq \theta_2$, con lo que se tiene el resultado.
- Si $E_2 = [y_1 : y_2]$, solamente puede aplicarse regla si θ_2 es variable de canal e y_1 se encuentra ligada a una expresión que no es un valor. En este caso la regla que se aplica es *stream* DEMAND. Se tiene que $dom(K^{(2,2)}) \subseteq \{\theta_2, y_1\}$. Por hipótesis, $x \neq \theta_2$. En cuanto a y_1, si es distinta de x se tiene el resultado. En el caso de que $x = y_1$, entonces en ambos casos la modificación de su ligadura es una activación, con lo que también se tiene el resultado.

- BLACKHOLE: entonces $K^{(1,2)} = \{\theta_1 \overset{B}{\mapsto} \theta_1\}$, y por el Lema A.36 la propiedad se cumple.

- LET: entonces $K^{(1,2)} = \{y^i \overset{I}{\mapsto} E^i\}_{i=1}^n \cup \{\theta_1 \overset{A}{\mapsto} E'\}$. Todas las variables x^i son nuevas, de manera que no pueden pertenecer al conjunto $K^{(2,2)}$, y el Lema A.36 garantiza que la ligadura de θ_1 no puede ser modificada por la evolución de θ_2.

- β-REDUCTION: entonces $K^{(1,2)} = \{\theta_1 \overset{A}{\mapsto} E'[y/x]\}$. Así $\nexists \theta_1' \overset{IB}{\mapsto} E' \in K^{(1,2)}$, y el Lema A.36 asegura que la propiedad se verifica.

- APP-DEMAND: en este caso $E_1 = x\,y$, siendo $K^{(1,2)} \subseteq \{x \overset{A}{\mapsto} E', \theta_1 \overset{B}{\mapsto} x\,y\}$. Igual que sucedía en el caso de la regla DEMAND, sólo tenemos que preocuparnos de lo que pueda suceder con x, y en particular, el caso en que x estuviera inactiva y ahora se hubiera activado, de donde se deduce que $x \neq \theta_2$. Un razonamiento análogo al aplicado para la regla DEMAND conduce al resultado.

- Λ-DEMAND: en este caso $E_1 = x\,y$, siendo $K^{(1,2)} \subseteq \{x \overset{A}{\mapsto} E', \theta_1 \overset{B}{\mapsto} x\,y\}$. Del mismo modo que sucedía en el caso de la regla DEMAND, sólo hay que tener en cuenta lo que pueda suceder con x, y en particular, el caso en que x estuviera inactiva y ahora se hubiera activado, de donde se deduce que $x \neq \theta_2$. Un razonamiento análogo al aplicado para la regla DEMAND conduce al resultado.

- B-REDUCTION EMPTY LIST: tras aplicar esta regla, $\nexists \theta_1' \overset{IB}{\mapsto} \in K^{(1,2)}$, y $K^{(1,2)} = \{\theta_1 \overset{A}{\mapsto} E'\}$, y el Lema A.36 asegura la propiedad.

- B-REDUCTION NON-EMPTY LIST: la aplicación de esta regla hace que $\nexists \theta_1' \overset{IB}{\mapsto} \in K^{(1,2)}$, y que $K^{(1,2)}$ sea $\{\theta_1 \overset{A}{\mapsto} E'\}$, y por el Lema A.36 se tiene la propiedad.

- *stream* DEMAND: en este caso $E_1 = [x : y]$, siendo $K^{(1,2)} \subseteq \{x \overset{A}{\mapsto} E', \theta_1 \overset{B}{\mapsto} [x : y]\}$. Del mismo modo que sucedía en el caso de la regla DEMAND, sólo hay que centrarse en lo que pueda suceder con x, y en particular, el caso en que x estuviera inactiva y ahora se hubiera activado, de donde se deduce que $x \neq \theta_2$. Un razonamiento análogo al aplicado para la regla DEMAND asegura el resultado.

- LEFT/RIGHT MERGE: entonces $\nexists \theta_1' \overset{IB}{\mapsto} \in K^{(1,2)}$, y $dom(K^{(1,2)}) = \theta_1, z$, donde z es una variable fresca, con lo que el Lema A.36 garantiza el resultado.

- MERGE TERMINATION: en este caso $K^{(1,2)} = \{\theta_1 \overset{A}{\mapsto} \texttt{nil}\}$, con lo que no contiene ligaduras ni bloqueaas ni inactivas, y por el Lema A.36 se tiene el resultado.

- BLOCKING MERGE: en todos los casos $dom(K^{(1,2)}) = \{\theta_1\}$, y de nuevo el Lema A.36 garantiza el resultado.

- CHANNEL CREATION: en este caso $dom(K^{(1,2)}) = \{\theta_1, x', y'\}$, donde x' e y' son frescas, garantizando el Lema A.36 la propiedad.

- CHANNEL NAME DEMAND: en este caso $E_1 = x\,!\,y_1\ \texttt{par}\ y_2$, siendo $K^{(1,2)} \subseteq \{x \overset{A}{\mapsto} E', \theta_1 \overset{B}{\mapsto} x\,!\,y_1\ \texttt{par}\ y_2\}$. Del mismo modo que sucedía en el caso de la regla DEMAND, sólo hay que centrarse en lo que pueda suceder con x, y en particular, el caso en que x estuviera inactiva y ahora se hubiera activado, de donde se deduce que $x \neq \theta_2$. Un razonamiento análogo al aplicado para la regla DEMAND asegura el resultado.

Analizados todos los casos, se garantiza la no interferencia entre la evolución de distintas ligaduras activas.

c.q.d.

A.15. Demostración de la Proposición 6.25

Proposición 6.25 *Sean $E_0 \notin Val$ una expresión cerrada bien formada, y $\langle p_0, \{main \overset{A}{\mapsto} E_0\}\rangle \Longrightarrow^+ (S, \langle p, H + \{\theta \overset{A}{\mapsto} E\}\rangle)$ una secuencia de configuraciones con al menos un paso. Entonces, si E no es ni una $\texttt{\#}$-expresión ni $z\,!\,y_1\,\texttt{par}\,y_2$ tal que $z \overset{I}{\mapsto} \texttt{ch}(c, x) \in H + \{\theta \overset{A}{\mapsto} E\}$, existirá H' tal que $H : \theta \overset{A}{\mapsto} E \longrightarrow H'$.*

Demostración P.6.25

Por inducción estructural sobre E.

- $E = \theta'$: como existen dos tipos de variables, hemos de distinguir dos casos:
  - $\theta' = x$: entonces tendremos $x \overset{\alpha}{\mapsto} E_1$. Si $E_1 = W$, entonces $\alpha = I$ pues el valor habrá sido obtenido en el paso global anterior y la ligadura habrá sido desactivada. Por ello, es posible aplicar la regla VALUE. Si E_1 no es un valor *whnf*, entonces se aplica la regla DEMAND. Finalmente, en el supuesto de que $\theta = \theta' = x$, se aplica a esta ligadura la regla BLACKHOLE.
  - $\theta' = ch$: este supuesto no es posible porque una ligadura que asocia una variable con un canal solamente puede ser una ligadura-esperando-canal (ver Notación 6.3), y, por ende, bloqueada y no activa.

- Si E es una abstracción o la lista vacía sucede lo mismo que cuando se suponía era un canal: no puede existir una ligadura del tipo $\theta \overset{A}{\mapsto} E$ (para una mayor aclaración véase el Ejemplo 6.19).

- $E = x\,y$: se contemplan dos posibilidades:
  - Que $x \overset{IAB}{\mapsto} E'$ y E' no sea una abstracción, en cuyo caso se demanda la evaluación de x, es decir, se aplica la regla APP-DEMAND.
  - Que $x \overset{I}{\mapsto} \backslash x.E_1$, con lo que se puede aplicar β-REDUCTION.
  - Que $x \overset{I}{\mapsto} \Lambda[x_1 : x_2].E_1 \| E_2$. Caben entonces tres casos:
    - Que $y \overset{I}{\mapsto} \texttt{nil}$, en cuyo caso se aplica la regla B-REDUCTION EMPTY LIST.
    - Que $y \overset{I}{\mapsto} [x_1 : x_2]$, aplicándose entonces la regla B-REDUCTION NON-EMPTY LIST.
    - Que $y \overset{I}{\mapsto} E'$ y $E' \notin \mathit{List}$, procediéndose entonces a aplicar la regla Λ-DEMAND.

- $E = \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E'$: se puede aplicar la regla LET.

- $E = \texttt{new}(y, x)E$: es aplicable la regla CHANNEL CREATION.

- $E = z\,!\,y_1\,\texttt{par}\,y_2$: la restricción del enunciado deja solamente cabida a que $z \overset{\alpha}{\mapsto} E \in H + \{\theta \overset{A}{\mapsto} z\,!\,y_1\,\texttt{par}\,y_2\}$ y $E \notin \mathit{Val}$, siendo entonces posible aplicar la regla CHANNEL NAME DEMAND.

- $E = x_1 \bowtie x_2$: la siguiente tabla muestra todos los posibles casos (donde $H'' = H' + \{\theta \overset{A}{\mapsto} E\}$ lm es una abreviatura de LEFT MERGE, rm de RIGHT MERGE, mt de

MERGE TERMINATION y bm de BLOCKING MERGE).

		x_2			
		$[y_1' : y_2']\wedge$ $y_1' \overset{I}{\mapsto} W''\wedge$ $\mathsf{nf}(W'', H'') = \emptyset$	$[y_1' : y_2']\wedge$ $y_1' \overset{\alpha}{\mapsto} E''\wedge$ $(E'' \notin Val \vee$ $(E'' \in Val\wedge$ $\mathsf{nf}(W'', H'') \neq \emptyset))$	`nil`	$E'' \notin List$
x_1	$[y_1 : y_2]\wedge$ $y_1 \overset{I}{\mapsto} W'\wedge$ $\mathsf{nf}(W', H'') = \emptyset$	lm/rm	lm	lm	lm
	$[y_1 : y_2]\wedge$ $y_1 \overset{\alpha}{\mapsto} E'\wedge$ $(E' \notin Val \vee$ $(E' \in Val\wedge$ $\mathsf{nf}(W', H'') \neq \emptyset))$	rm	bm 4	bm 2	bm 2
	`nil`	rm	bm 3	mt	bm 1
	$E' \notin List$	rm	bm 3	bm 1	bm 1

- $E = [x_1 : x_2]$: solamente cabe esta posibilidad si $\theta = ch \wedge x_1 \overset{\alpha}{\mapsto} E \wedge E \notin Val$, aplicándose entonces la regla *stream* DEMAND.

Examinados todos los casos posibles de expresión hemos visto que en todos ellos es posible aplicar una regla local.

c.q.d.

APÉNDICE B

Glosario

Las definiciones incluidas en este glosario han sido tomadas de [How93], salvo la de currificación, extraída de [Sto77], y la de redex, que se encuentra en [HM99].

Call-by-name: orden normal de reducción, i.e. se reduce el redex más a la izquierda y más externo.

Call-by-need: estrategia de reducción que retrasa la evaluación de los argumentos de una función hasta que son necesarios por ser argumentos de una función primitiva o de un condicional. Este tipo de reducción es parte de la evaluación perezosa. El término fue acuñado por Christopher P. Wadsworth en su tesis doctoral [Wad71].

Call-by-value: estrategia de reducción según la cual los argumentos son evaluados antes de que se entre en la función o el procedimiento. Solamente se pasan los valores de los parámetros; y los cambios sobre los argumentos dentro de la llamada del procedimiento no tienen efecto sobre los parámetros reales.

Continuation passing style: es un estilo de programación en el cual cada función, f, toma un argumento extra, θ, denominado *continuación*. Siempre que f esté en disposición de devolver un resultado v a su invocador, no realizará esta devolución, y, en su lugar, devolverá el resultado de aplicar la continuación θ a v. De este modo, la continuación representa el resto del cómputo.

Currificación: transformación que hace corresponder a cualquier función de dos (o más) argumentos otra equivalente que toma los argumentos de uno en uno. Así, a $f(x, y)$

le corresponde $f'(x)(y)$. f' se denomina versión currificada de f; es una función de un argumento, x; el resultado, $f'(x)$, es también una función de un argumento, y, cuyo resultado, $f'(x)(y)$, es igual a la original $f(x, y)$.

Estrategia de reducción: algoritmo para decidir cuál(es) redex(es) es(son) el(los) siguientes en ser reducido(s). Cuando se está tratando con funciones o valores recursivos, diferentes estrategias conducen a diferentes propiedades de terminación.

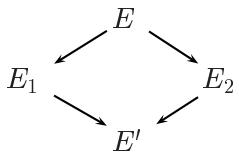
Evaluación impaciente: se dice de toda estrategia de evaluación donde la evaluación de alguno o todos los argumentos de una función empieza antes de que el valor sea requerido. Un ejemplo típico es la evaluación *call-by-value*.

Evaluación perezosa: estrategia de evaluación que combina el orden normal de evaluación con la actualización. Bajo el orden normal de evaluación (se reduce el redex más externo, o evaluación *call-by-name*) una expresión es evaluada solamente cuando su valor es necesitado para que el programa devuelva su valor. Actualización quiere decir que si el valor de una expresión es necesitado en más de una ocasión, el resultado de la primera evaluación se recuerda y las demandas posteriores de dicho valor devolverán este valor recordado sin más evaluación [How93].

Forma normal: una expresión está en forma normal si su nivel superior ni es un redex ni una λ-abstracción con un cuerpo reducible.

Pereza completa: transformación del programa que optimiza la evaluación perezosa asegurando que todas aquellas subexpresiones en el cuerpo de una función que no dependan de los argumentos de dicha función, sean evaluadas a lo sumo una vez.

Propiedad de Church-Rosser: establece que si una expresión E puede ser reducida en cero o más pasos de reducción a dos expresiones distintas E_1 y E_2, entonces existe otra expresión E' a la que tanto E_1 como E_2 pueden ser reducidas.



Esto implica que existe una única forma normal para cualquier expresión, ya que E_1 y E_2 no pueden ser formas normales diferentes porque la propiedad establece que ninguna de ellas puede ser reducida a otra expresión y las formas normales son irreducibles por definición. Sin embargo, esto no implica que una forma normal sea alcanzable, solamente que si la reducción termina ésta alcanzará una única forma normal.

Redex(es): una expresión reducible (*reducible expression*), i.e. una expresión que puede ser transformada a alguna forma normal, usando las reglas de reducción del λ-cálculo.

Transparencia referencial: una expresión E es referencialmente transparente si cualquier subexpresión suya y su valor, i.e. el resultado de evaluarla, pueden ser intercambiados sin cambiar el valor de E. Este no es el caso si el valor de una expresión depende de un estado global que puede cambiar dicho valor. El ejemplo más común de cambio del estado global es la asignación a una variable global (efecto lateral). Los lenguajes funcionales puros posibilitan la transparencia referencial prohibiendo la asignación a variables globales. Cada expresión es o una constante o una aplicación funcional cuya evaluación no tiene efectos laterales, solamente devuelve un valor que depende únicamente de la definición de la función y de los valores de sus argumentos. Los programas referencialmente transparentes son más susceptibles de ser tratados con métodos formales y es más sencillo razonar sobre ellos, pues el significado de una expresión depende solamente del significado de sus subexpresiones y no del orden de evaluación de los efectos laterales de otras expresiones.

whnf: una expresión está en forma normal débil de cabeza (*weak head normal form*) si es una forma normal o cualquier λ-abstracción, i.e. su nivel superior no es un redex. El término fue acuñado por Simon Peyton Jones.

Bibliografía

[AAA+95] S. Aditya, Arvind, L. Augustsson, J. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In P. Hudak, editor, *Haskell Workshop*, pages 35–49, La Jolla, Cambridge, MA, USA. Technical Report YALEU/DCS/RR-1075, June 1995.

[Abd73] S. K. Abdali. A simple lambda-calculus model of programming languages. AEC R & D C00-3077-28, New York University, 1973.

[Abr90] S. Abramsky. *Research Topics in Functional Programming*, chapter 4: The lazy lambda calculus, pages 65–117. Ed. D. A. Turner. Addison Wesley, 1990.

[Bak00] C. Baker-Finch. An abstract machine for parallel lazy evaluation. In *Trends in Functional Programming (Selected papers of the First Scottish Functional Programming Workshop)*, volume 1, pages 153–161. Intellect, 2000.

[BHA85] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *Programs as data objects*, pages 42–62, Copenhagen, Denmark, 1985. Springer.

[BK84] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. Technical Report CS-R8420, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1984.

[BKHT99] C. Baker-Finch, D. King, J. Hall, and P. Trinder. An operational semantics for parallel call-by-need. Technical Report 99/1, Faculty of Mathematics and Computing, The Open University, 1999.

[BKL+97] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM – the DistRibuted Eden Abstract Machine. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages, (IFL'97 selected papers)*, pages 250–269. LNCS 1467, Springer, 1997.

[BKT00] C. Baker-Finch, D. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173, Montreal, Canada, September 2000.

[BLO94] K. Bohlmann, R. Loogen, and Y. Ortega-Mallén. Towards a functional process calculus. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *1994 Joint Conference on Declarative Programming, GULP-PRODE'94*, pages 234–250, Peñiscola, Spain, Sept. 1994. Universitat Politècnica de València.

[BLO96] S. Breitinger, R. Loogen, and Y. Ortega-Mallén. Towards a declarative language for concurrent and parallel programming. In D. N. Turner, editor, *Glasgow Workshop on Functional Programming 1995*. Electronic Workshops in Computing, Springer, 1996.

[BLOP96a] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden – the paradise of functional-concurrent programming. In *European Conference on Parallel Processing, EUROPAR'96*, pages 710–713. LNCS 1123, Springer, 1996.

[BLOP96b] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report 96/10, Reihe Informatik, FB Mathematik, Philipps-Universität Marburg, Germany, URL `http://www.mathematik.uni-marburg.de/inf/eden/index.html`, 1996.

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, 1997.

[Cas98] John L. Casti. *El Quinteto de Cambridge*. Taurus, 1998.

[CC94] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[CDE+00] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer y J. F. Quesada N. Martí-Oliet. A maude tutorial. Computer Science Laboratory, SRI International. URL `http://maude.csl.sri.com/tutorial`, 2000. Tutorial distribuido como documentación del sistema Maude.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[DB97] M. Debbabi and D. Bolignano. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 6: A semantic theory for ML higher-order concurrency primitives, pages 145–184. Monographs in Computer Science. Ed. F. Nielson. Springer, 1997.

[DFH+93] J. Darlington, A. J. Field, P. J. Harrison, P. H. Kelly, D. W. Sharp, and Q. Wu. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe, PARLE'93*, pages 146–160. Springer, 1993.

[DGH+82] J. Darlington, J. V. Guttag, P. Henderson, J. H. Morris, J. E. Stoy, G. J. Sussman, P. C. Treleaven, D. A. Turner, J. H. Williams, and D. S. Wise. *Functional Programming and its Applications*. Eds. J. Darlington and P. Henderson and D. A. Turner. Cambridge University Press, 1982.

[Dij60] E. W. Dijkstra. Recursive programming. *Numerische Mathematik*, 2:312–318, 1960.

[EP02] A. de la Encina and R. Peña. Proving the correctness of the STG machine. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages, (IFL'01 selected papers)*, pages 88–104. LNCS 2312, Springer, 2002.

[EP03] A. de la Encina and R. Peña. Formally deriving an STG machine. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming, PPDP'03*, pages 102–112. ACM Press, 2003.

[Fer96] W. Ferreira. *Semantic Theories for Concurrent ML*. PhD thesis, University of Sussex (Dept. of Cognitive and Computing Sciences), 1996.

[FH99] W. Ferreira and M. Hennessy. A behavioural theory of first-order CML. *Theoretical Computer Science*, 216:55–107, 1999.

[Fis93] M. J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation: An International Journal*, 6(3-4):257–286, November 1993.

[GKK+93] A. Giacalone, F. Knabe, A. Kramer, T. Kuo, L. Leth, S. Prasad, and B. Thomsen. Facile antigua release programming guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, 1993.

[Gla87] R. J. van Glabbeek. Bounded nondeterminism and the approximation induction principle in process algebra. In G. Goos and J. Hartmanis, editors, *Proceedings of the Fourth Annual Symposium on Theoretical Aspects of Computer Science, STACS'87*, pages 336–347. LNCS 247, Springer, 1987.

[GMP90] A. Giacalone, P. Mishra, and S. Prasad. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *Proceedings of International Colloquium on Automata,Languages, and Programming '90*, pages 765–780. LNCS 443, Springer, 1990.

[HBTK99] J. G. Hall, C. Baker-Finch, P. Trinder, and D. J. King. Towards an operational semantics for a parallel non-strict functional language. In *Proceedings of the 10th International Workshop on Implementation of Functional Languages, (IFL'98 selected papers)*, pages 55–72. LNCS 1595, Springer, 1999.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation.* Prentice Hall International Series in Computer Science, 1980.

[Hen82] P. Henderson. *Functional Programming and its Applications*, chapter Purely Functional Operating Systems, pages 177–192. Eds. J. Darlington and P. Henderson and D. A. Turner. Cambridge University Press, 1982.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hen90] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* John Wiley and Sons, 1990.

[Hid99] M. Hidalgo-Herrero. Semánticas para lenguajes funcionales concurrentes y paralelos. Master's thesis, Universidad Complutense de Madrid (Departamento de Sistemas Informáticos y Programación), 1999.

[HM99] K. Hammond and G. Michaelson (editors). *Research Directions in Parallel Functional Programming.* Springer, 1999.

[HO00] M. Hidalgo-Herrero and Y. Ortega-Mallén. A distributed operational semantics for a parallel functional language. In *Trends in Functional Programming (Selected papers of the Second Scottish Functional Programming Workshop)*, volume 2, pages 89–102. Intellect, 2000.

[HO01a] M. Hidalgo-Herrero and Y. Ortega-Mallén. A distributed operational semantics for Eden. In *IX Jornadas de Concurrencia*, pages 129–144. Universitat Ramon Llull, 2001.

[HO01b] M. Hidalgo-Herrero and Y. Ortega-Mallén. Towards a denotational semantics for Eden. In *Draft Proceedings of the Third Scottish Functional Programming Workshop*, pages 45–54. University of Stirling, Scotland, 2001.

[HO02a] M. Hidalgo-Herrero and Y. Ortega-Mallén. A continuation-based model for a distributed functional setting. In *X Jornadas de Concurrencia*, pages 87–101. Universidad de Zaragoza, Spain, 2002.

[HO02b] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[HO02c] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. In *Proceedings of the Third International Workshop on Constructive Methods for Parallel Programming, CMPP'02*, pages 63–79. Technische Universitaet Berlin, Germany, 2002.

[HO02d] M. Hidalgo-Herrero and Y. Ortega-Mallén. Tailoring laziness for a distributed setting. A denotational point of view. Presented in the *Eighteenth Workshop on the Mathematical Foundations of Programming Semantics*, MFPS XVIII, 2002.

[HO03a] M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *Draft Proceedings of the 15th International Workshop on Implementation of Functional Languages, IFL'03*, pages 129–143. School of Mathematical and Computer Sciences, Heriot Watt University and Formal Aspects of Computing Science Special Interest Group, British Computer Society, 2003.

[HO03b] M. Hidalgo-Herrero and Y. Ortega-Mallén. Continuation semantics for parallel Haskell dialects. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems, (APLAS'03)*, pages 303–321. LNCS 2895, Springer, 2003.

[Hoa73] C. A. R. Hoare. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[How93] D. Howe. FOLDOC: Free On-Line Dictionary of Computing. URL `http://www.foldoc.org`, Imperial College Department of Computing, 1993.

[HPR00] F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In *Trends in Functional Programming (Selected papers of the First Scottish Functional Programming Workshop)*, volume 1, pages 11–19. Intellect, 2000.

[HR00] K. Hammond and A. J. Rebón Portillo. HaskSkel: Algorithmic skeletons in Haskell. In *Proceedings of the 11th International Workshop on Implementation of Functional Languages, (IFL'99 selected papers)*, pages 181–198. LNCS 1868, Springer, 2000.

[Hsu04] C. Hsu. El Colón que vino de China. *EPS - El País Semanal*, 1436:27–30, 4th april 2004. Traducción del publicado en US News & World Report en 2003.

[Jos89] M. B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68:105–111, 1989.

[Kel89] P. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Pitman, 1989.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[Lan65a] P. J. Landin. A correspondence between ALGOL 60 and Chuch's lambda-notation. *Communications of the ACM*, 8(2-3):89–101 and 158–165, February-March 1965.

[Lan65b] P. J. Landin. A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research, August 1965.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.

[Loi96] H. W. Loidl. *GranSim user's guide.* GRASP/AQUA Proyect, Glasgow University, 1996.

[Loo99] R. Loogen. *Research Directions in Parallel Functional Programming*, chapter 3: Programming Language Constructs, pages 63–92. Eds. K. Hammond and G. Michaelson. Springer, 1999.

[LOP+02] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 4: Parallelism Abstractions in Eden, pages 95–128. Eds. F. A. Rabhi and S. Gorlatch. Springer, 2002.

[MAE+62] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. J. Hart, and M. I. Levin. *The LISP 1.5 Programmer's Manual.* MIT Press, 1962.

[Maz71] A. W. Mazurkievicz. Proving algorithms by tail functions. *Information and Control*, 18(3):220–226, April 1971.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mol96] M. Moliner. *María Moliner - Diccionario del Uso del Español - Versión Electrónica, 1.0.* Editorial Gredos, 1996. (A partir del "Diccionario de Uso del Español" de María Moliner, 1ª edición).

[Mon03] R. Montero. La belleza. *EPS - El País Semanal*, 1415:190, 9th november 2003.

[Mor82] J. H. Morris. *Functional Programming and its Applications*, chapter Real Programming in Functional Languages, pages 129–176. Eds. J. Darlington and P. Henderson and D. A. Turner. Cambridge University Press, 1982.

[Mor93] F. L. Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3-4):249–256, November 1993. Original manuscript (unpublished): November 1970.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, 1990.

[NA01] R. S. Nikhil and Arvind. *Implicit Parallel Programing in pH.* Academic Press, 2001.

[Nau63] P. Naur. The design of the GIER ALGOL compiler, part i. *BIT*, pages 124–140, 1963. Reprinted in Goodman, R., editor, *Annual Review in Automatic Programming*, Vol. 4, pages 49-85, Pergamon Press, Oxford, 1994.

[NSEP91] E. G. Nöcker, J. E. W. Smetsers, M. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In E. H. Aarts, J. van Leeuwen, and M. Rem, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE) II*, pages 202–219. LNCS 505, Springer, 1991.

[Oak93] Oak Ridge National Laboratory, University of Tennessee. *Parallel Virtual Machine Reference Manual Version 3.2*, 1993.

[OH86] E. R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

[Pey87] S. Peyton Jones. *Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pey03] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report.* Cambridge University Press, 2003.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Symposium on Principles of Programming Languages, POPL'96*, pages 295–308. ACM Press, January 1996.

[PH99] S. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98. URL `http://www.haskell.org`, February 1999.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[PPRS00] C. Pareja, R. Peña, F. Rubio, and C. Segura. Optimizing Eden by Transformation. In *Trends in Functional Programming (Selected papers of the 2nd Scottish Functional Programming Workshop)*, volume 2, pages 13–26. Intellect, 2000.

[PR97] P. Panangaden and J. Reppy. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 2: The Essence of Concurrent ML, pages 5–30. Monographs in Computer Science. Ed. F. Nielson. Springer, 1997.

[PRS02] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In *Trends in Functional Programming (Selected papers of the Third Scottish Functional Programming Workshop)*, volume 3, pages 51–62. Intellect, 2002.

[Rea95] Real Academia Española. *Diccionario de la Lengua Española - Edición Electrónica, 21.1.0.* Espasa Calpe, S.A., 1995. (A partir del "Diccionario de la Lengua Española" de la Real Academia Española, edición 1992).

[Rep92] J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University (Department of Computer Science), 1992.

[Rep93] J. H. Reppy. Concurrent ML: Design, aplication and semantics. In *Proceedings of Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. LNCS 693, Springer, 1993.

[Rey72] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 717–740. ACM Press, 1972.

[Rey93] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation: An International Journal*, 6(3-4):233–247, November 1993.

[Rey98] J. C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[Rub99] F. Rubio. Programación funcional-paralela eficiente. Master's thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 1999.

[Rub01] F. Rubio. *Programación funcional paralela eficiente en Eden.* PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[SP03] C. Segura and R. Peña. Correctness of non-determinism analyses in a parallel-functional language. Technical Report 131-03, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2003.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[Sto82] J. E. Stoy. *Functional Programming and its Applications*, chapter Some Mathematical Aspects of Functional Programming, pages 217–252. Eds. J. Darlington and P. Henderson and D. A. Turner. Cambridge University Press, 1982.

[SW00] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computarion*, 13:135–152, 2000. Original Manuscript: Technical Monograph PRG-11, Oxford University Computing Laboratory, January, 1974.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

[Ten76] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, August 1976.

[THLP98] P. W. Trinder, K. H. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Funcional Programming*, 8(1):23–60, January 1998.

[THM+96] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation, PLDI'96*, pages 78–88, Philadephia, USA, May 1996. ACM Press.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University (Programming Research Group), 1971.

[Wij66] A. van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24, Amsterdam, 1966. North-Holland.

[Wik94] C. Wikström. Distributed programming in Erlang. In Hoon Hong, editor, *PASCO'94: First International Symposium on Parallel Symbolic Computation*, pages 412–421. World Scientific Publishing Company, September 1994.