Centro de Investigación y Estudios
Avanzados
del Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica
Sección de Computación

Diseño e implementación de una máquina virtual para el cálculo de ambientes.

Tesis que presenta

Luis Rodrigo Gallardo Cruz

Para obtener el grado de

Maestro en Ciencias
En la especialidad de Ingeniería Eléctrica con opción en Computación

Directores de tesis:
Dr. José Oscar Olmedo Aguirre
Dr. Alberto Oliart Ros

México, D.F. Septiembre, 2005

II

Agradecimientos

El autor de este trabajo desea agradecer su apoyo a las siguientes personas e instituciones:

A CINVESTAV.

A CONACyT, por el apoyo económico recibido a través de la beca 165921.

A LANIA, por el uso de sus instalaciones.

A mis asesores, por el tiempo invertido en este trabajo.

Al Dr. Carlos Artemio Coello.

A mi familia.

A Adriana.

Este trabajo fue realizado como parte del proyecto *Ambiente de Programación Distribuida y Móvil* bajo financiamiento de CONACyT (Ref: 37476-A).

Resumen

El cómputo móvil estudia los problemas peculiares al uso de dispositivos de cómputo móviles, conectados entre sí por redes inalámbricas. A pesar de las similitudes con el cómputo distribuido la investigación en el área ha mostrado que existen diferencias sustanciales entre las dos disciplinas. Como resultado, las herramientas teóricas y prácticas del cómputo distribuido no pueden ser usadas para el cómputo móvil.

Hay una gran cantidad de trabajo realizado en el desarrollo de modelos formales para el cómputo móvil, incluyendo en particular al cálculo de ambientes, pero poco de este trabajo se ha trasladado hacia el desarrollo de lenguajes de programación para cómputo móvil, lo cual inhibe la experimentación y el uso práctico.

Este trabajo describe el diseño e implementación de una máquina abstracta para ejecutar el cálculo de ambientes, a la que hemos llamado MAC. La implementación de esta máquina incluye una herramienta que ayuda a visualizar los pasos de reducción en el cálculo de ambientes, para propósitos educativos o de experimentación. La contribución principal de este trabajo consiste en que la disponibilidad de esta máquina ayudará en la búsqueda de lenguajes de programación adecuados para el cómputo móvil.

Abstract

Mobile computing studies the problems associated with the use of mobile computing devices interconnected by wireless networks. In spite of similarities with distributed computing, research in the area has shown that there are substantial differences between these two disciplines. As a result, theoretical and practical tools from distributed computing can not be used in mobile computing.

There is a great body of work in the area of formal models for mobile computing, including in particular the Ambient Calculus, but little of this work has been translated into the development of programming languages for mobile computing, which inhibits experimentation and practical use.

This work describes the design and implementation of an abstract machine for execution of the Ambient Calculus, which we call MAC. The implementation of the machine includes a tool to help visualize reduction steps in the ambient calculus, for experimental or educational purposes. The contribution of this work is that the availability of this machine will aid in the search for adequate programming languages for mobile computing.

Índice general

Notación

$!$	Operador de replicación
$\mathcal{N}$	El conjunto de nombres de ambientes
$(\nu\, n)P$	Introducción de n como un nuevo nombre privado al proceso P
$\mathcal{P}$	El conjunto de los procesos de la máquina virtual
$\mid$	Composición paralela de procesos
$(n).P$	Un proceso que recibe una comunicación, la asigna al nombre n y continúa como el proceso P
$\langle M \rangle$	Un proceso que emite M sobre el "canal" anónimo implícito a un ambiente
Σ	Un estado de la máquina virtual
$A \ominus B$	El ambiente que resulta de eliminar a B como hijo de A
$A \oplus B$	El ambiente que resulta de añadir a B como hijo de A
$H(c)$	Si H es un *heap*, la lista de procesos asociada a c
$M.P$	Proceso que ejecuta la capacidad M, y luego se comporta como P
$M[P]$	Un ambiente de nombre M y contenido P
P, Q, ...	Procesos en un cálculo de procesos
T	Una función que traduce estados de la máquina virtual al cálculo de ambientes

Top El nombre especial del ambiente de nivel superior en la máquina virtual

0 El proceso nulo

heap Una colección de listas de espera de procesos que aún no se pueden ejecutar

Introducción

El cómputo móvil es una disciplina relativamente nueva, pero que promete aumentar de importancia en el futuro cercano debido a la presencia cada vez más común de dispositivos móviles. La naturaleza de estos dispositivos es el origen del nombre: nodos pequeños, generalmente con capacidades de cómputo limitadas, con conexiones inalámbricas y que se desplazan físicamente entre una red y otra.

La tendencia general en los sistemas de cómputo distribuido es ocultar a los programas que se ejecutan en ellos la existencia de la red. En efecto, su objetivo último es presentar la ilusión de ser una única máquina paralela, de preferencia una semánticamente equivalente a una máquina secuencial, pero más rápida.

Esta solución no es factible en el ambiente del cómputo móvil. La suposición básica de una red de alcance global, mantenida en buena medida por medio de comunicaciones inalámbricas que unen entre sí a redes tecnológica y administrativamente independientes, invalida muchas de las suposiciones básicas que permiten el modelo tradicional del cómputo distribuido.

Algunas de las diferencias más notables son [7, 34]:

- En una red de área local se supone que ésta no fallará durante la ejecución de un programa. En general los errores son pocos y espaciados y pueden ser manejados por el ambiente de cómputo sin necesidad de que los programas se enteren. En cambio, en una red móvil los errores son más frecuentes, y es posible que los programas necesiten tomar acción explícita en presencia de éstos.

- La extensión geográfica de estas redes, por citar sólo un factor, impone límites en la velocidad de comunicación, que son insuperables sin

importar el grado de avance tecnológico. El desempeño futuro de la red no puede ser pronosticado a partir de lo observado en el presente.

- La configuración de la red es dinámica. Nodos y subredes enteras pueden añadirse o desaparecer sin previo aviso. Además, debido al punto anterior, estos sucesos son indistinguibles de fallas transitorias. El ancho de banda disponible puede variar de forma dramática e impredecible.

- La red consta de dominios administrativos separados, que de hecho desconfían unos de otros. Por este motivo la comunicación entre ellos está severamente limitada y se requieren permisos especiales para migrar de uno a otro. El cómputo móvil es inherentemente peligroso, puesto que los nodos están expuestos a riesgos, como el que un equipo pueda perderse, sufrir daño físico o ser robado.

La realización plena del potencial del cómputo móvil depende de tener herramientas que nos permitan atacar directamente las limitaciones específicas del medio. La investigación ha producido modelos teóricos que prometen un mejor entendimiento de los fenómenos relevantes. Es necesario ahora probar estos modelos por medio de la experimentación, para conocer sus fallos y aciertos desde el punto de vista de la ingeniería de software. Un modelo formal para el cómputo móvil debe ser expresivo, pero también fácil de entender y de usar. Además, para que estas cualidades se reflejen en la práctica, debe haber herramientas como compiladores o intérpretes al alcance de los posibles usuarios.

El presente trabajo presenta el diseño e implementación de una máquina virtual para ejecutar uno de estos modelos formales, el Cálculo de Ambientes, propuesto por Gordon y Cardelli en 1998 [12]. A partir de esta base será posible experimentar con este modelo y con otros relacionados que han sido presentados en la literatura, ayudando a confirmar la utilidad práctica de éstos.

Esta tesis está estructurada como sigue:

En el capítulo 1 se presentan brevemente varios modelos formales desarrollados tanto para el cómputo distribuido como para el cómputo móvil. Se da un resumen de sus características y se justifica la elección del cálculo

de ambientes como el modelo sobre el que se basa el resto del trabajo. Se da también un breve resumen de lenguajes de programación basados en estos modelos.

El capítulo 2 presenta la definición formal y una explicación intuitiva del significado del cálculo de ambientes.

El capítulo 3 presenta el diseño de la máquina abstracta. Se define su semántica operacional y se demuestra un resultado de corrección. El capítulo 4 presenta los aspectos principales de la implementación.

Finalmente, en el capítulo 5 se da un resumen de las conclusiones. Se justifica la viabilidad de la máquina propuesta y se delinean posibilidades de trabajo futuro.

Adicionalmente el apéndice A presenta ejemplos de la ejecución de la máquina virtual, el apéndice B contiene el código fuente completo y el apéndice C da las instrucciones para obtenerlo en formato electrónico y ejecutar la implementación.

Capítulo 1

Modelos teóricos para cómputo móvil

El problema de añadir movilidad a la programación ha sido estudiado por diversos grupos desde los años 80. Esta investigación ha producido diversos modelos, que enfatizan aspectos diferentes del problema, dependiendo de los intereses principales de cada grupo.

1.1. Modelos previos

Los antecedentes teóricos se encuentran en los modelos desarrollados para tratar los problemas de la concurrencia y el paralelismo. Los primeros modelos exitosos fueron CSP[1], propuesto por C.A.R. Hoare [21], y CCS[2], propuesto por Milner [26].

Ambos cálculos proponen la comunicación entre procesos concurrentes como noción primitiva de cómputo, tal como el cálculo λ propone la aplicación de funciones [32, 3, 15] o la máquina de Turing la modificación de posiciones de memoria [36, 35].

Estos modelos marcan el inicio del estudio formal de la concurrencia en sistemas de cómputo.

[1] *Communicating Sequential Proceses*
[2] *Calculus of Communicating Systems*

CSP tuvo aplicación práctica como la base para Occam, un lenguaje para cómputo paralelo [20, 23].

1.1.1. Cálculo π

CSP y CCS incluyen, además de los conceptos de procesos y canales, una gran cantidad de tipos de datos, que son requeridos para poder modelar el cómputo de funciones. Esto los hace más complicados de especificar y de tratar en forma algebraica.

Además, en ambos la lista de los canales a los que tiene acceso un proceso está determinada por su forma sintáctica y no puede variar a lo largo de su vida.

Tomando como inspiración el cálculo λ, que construye todos estos tipos de datos de forma interna, a partir de sólo 2 operaciones primitivas, se buscó obtener un modelo para la concurrencia que tuviera la misma economía y elegancia teórica. Este trabajo culminó con la creación del cálculo π, por Milner, Parrow y Walker [28].

El cálculo π es un cálculo de procesos en el que la operación primitiva fundamental es la comunicación. Ésta se realiza a través de canales con nombre en donde los datos transmitidos son los canales mismos.

Para describir el cálculo π usamos el estilo de la máquina química abstracta de Boudol [4].

Este estilo de descripción de un cálculo separa en dos clases las reglas necesarias para especificar su semántica operacional. Por un lado se tienen *reglas estructurales* que describen el comportamiento meramente sintáctico de la notación. Por ejemplo esta clase de reglas especifica cuáles operadores son conmutativos o asociativos.

Las reglas estructurales definen una relación de equivalencia entre las fórmulas del lenguaje. Sobre el cociente de esta relación de equivalencia se definen *reglas de reducción* que especifican el comportamiento semántico del sistema. De esta forma se separa el comportamiento de los objetos modelados por el cálculo de los artefactos introducidos por la notación.

Específicamente, sea $\mathcal{L}$ un lenguaje sobre un alfabeto dado. Definimos una relación de equivalencia $\equiv$ sobre $\mathcal{L}$ a la que llamamos *congruencia*

estructural. En el cociente $\mathcal{L}/\equiv$ se define una relación, denotada por $\longrightarrow$, que especifica las reglas atómicas de reducción de la semántica operacional y denotamos por $\longrightarrow^*$ a su cerradura simétrica transitiva. Dados dos términos $p, q \in \mathcal{L}$ decimos que p *reduce a* q si $[p] \longrightarrow^* [q]$.

Por poner un ejemplo, el cálculo π define un operador de composición paralela $|$ que representa la ejecución simultánea de dos procesos P y Q. Es claro que ésta es una operación simétrica respecto a ambos procesos. Sin la distinción entre reglas estructurales y reglas de reducción es necesario especificar que una reducción dada se puede aplicar tanto al primer proceso de $P|Q$ como al segundo por medio de reglas de reducción para cada caso. Con la distinción, se define una sola regla sobre el objeto abstracto del que tanto $P|Q$ como $Q|P$ son representaciones sintácticas válidas, lo que permite una mayor claridad en la exposición.

Definimos a continuación el cálculo π. Las reglas presentadas se tomaron de [27].

Sea $\mathcal{N}$ un conjunto infinito de nombres. Denotamos a los miembros de este conjunto por x, y, z y, ocasionalmente, por otras (a, b, ...).

El lenguaje del cálculo está dado por la gramática

$$\begin{aligned} P := \; & xy.P \quad | \\ & x(y).P \quad | \\ & 0 \quad | \\ & P|\,Q \quad | \\ & !P \quad | \\ & (y)P \end{aligned}$$

Informalmente, . representa composición secuencial y $|$ composición paralela. ! representa *replicación* y es más o menos equivalente a $P|\,P|\,P\ldots$, tantas veces como se desee. 0 es el proceso inactivo.

La primera y segunda regla modelan la comunicación. La primera significa "envía (el nombre) y sobre el canal x y después ejecuta P". La segunda significa "recibe (un nombre) z sobre x y entonces ejecuta $P\{y/z\}$", en donde esta última notación representa la sustitución de todas las ocurrencias de y por z en P.

La última regla restringe el alcance de un nombre y al proceso P. Tanto esta operación como $x(y).P$ acotan al nombre y, por lo que éste puede ser sustituido de forma consistente dentro del proceso P. Decimos que un nombre está libre en P si aparece en P pero no está vinculado.

Definimos la congruencia estructural como la mínima relación de equivalencia que cumple las siguientes ecuaciones:

1. $P \equiv Q$ si Q se obtiene de P sustituyendo los nombres vinculados. (La regla de conversión α del cálculo λ [3].)
2. $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.
3. $!P \equiv P \mid !P$.
4. $(x)0 \equiv 0$, $(x)(y)P \equiv (y)(x)P$.
5. $(x)(P \mid Q) \equiv P \mid (x)Q$ siempre que x no aparezca libre en P.

Las reglas de reducción son

COM: $x(y).P \mid \overline{x}z.Q \longrightarrow P\{y/z\} \mid Q$,

PAR: $P \longrightarrow P' \Longrightarrow P \mid Q \longrightarrow P' \mid Q$ y

RES: $P \longrightarrow P' \Longrightarrow (y)P \longrightarrow (y)P'$,

definidas, como se dijo antes, sobre las clases de equivalencia de la congruencia estructural.

Una de las características sobresalientes del cálculo es la *extrusión*. Definamos los procesos $P = \overline{y}x.P'$ y $Q = y(z)Q'$, suponiendo que x no aparece en Q' y consideremos las siguientes reducciones:

$$\begin{aligned}(x)(P \mid R) \mid Q &= (x)(\overline{y}x.P' \mid R) \mid y(z).Q' \\ &\equiv (x)(\overline{y}x.P' \mid R \mid y(z).Q') \\ &\longrightarrow (x)(P' \mid R \mid Q'\{z/x\}),\end{aligned}$$

donde la equivalencia se justifica porque x no aparece en Q.

En este caso, el alcance de x, que originalmente era un nombre privado a P y R, se extiende para abarcar a Q' después de la comunicación. El

cálculo modela adecuadamente la intuición de que el proceso que recibe un nombre puede operar con él como si lo tuviese desde el principio. Con este mecanismo en el cálculo π se resuelve un problema de CSP y CCS, los cuales requieren que el conjunto de procesos con los que otro puede comunicarse esté fijo desde el momento de su definición.

El cálculo π es un modelo de cómputo completo que puede representar todas las características necesarias para un lenguaje de programación completo. En particular, es posible codificar tipos de datos básicos como los enteros y estructuras de datos como, por ejemplo, arreglos.

Si bien no es un cálculo de orden superior, puesto que los procesos no son en sí mismos valores comunicables, sí es posible codificar este comportamiento transmitiendo el conjunto de canales sobre los que escucha el proceso.

Todas estas características son resumidas en un resultado fundamental: El cálculo π es capaz de codificar fielmente el cálculo λ [27], lo cual establece la equivalencia teórica entre ambos. Esto es relevante ya que el cálculo λ es uno de los modelos teóricos de computabilidad. Poder codificarlo usando el cálculo π significa que cualquier problema computable (es decir, cualquier problema factible de ser resuelto por una computadora) es expresable usando este último.

Debido a esto, el cálculo π es el modelo fundamental en el que se basa el estudio de la concurrencia.

Sin embargo, no es un modelo adecuado para estudiar el cómputo distribuido o el móvil. Esto se debe a que existen varios conceptos importantes para estos fenómenos que no tienen una representación natural. Específicamente, el cálculo π no tiene una noción natural de *posición* [29]. El concepto de movilidad que se puede codificar naturalmente en el cálculo π es una movilidad *lógica,* consistente en la modificación dinámica de los canales de comunicación disponibles a un proceso.

1.1.2. *Join calculus*

Aún en el cómputo distribuido, en el que la meta es esconder la existencia de 'lugares' diferentes, es necesario contar con un modelo que las tome

en cuenta explícitamente. Esto es así por dos razones fundamentales:

- Si cualquier proceso puede escuchar o emitir sobre cualquier canal, independientemente de su localización, es necesario tener un mecanismo de sincronización entre ellos. En el caso en que se ejecuta un proceso distribuido, todos los algoritmos conocidos enfrentan rápidamente un problema de escalabilidad.

- El cómputo distribuido debe tener en cuenta la posibilidad de que una localidad falle independientemente de las demás.

Atendiendo a estas consideraciones, Cédric Fournet y Georges Gonthier propusieron en 1996 el *Join Calculus* [19].

No proporcionamos la definición del cálculo, pues presenta varios detalles sutiles[3] y porque no es relevante para el resto del trabajo. Sin embargo, se presentan algunas de sus características relevantes, que sirvieron de inspiración para algunas decisiones de diseño. El lector interesado puede consultar [19].

El *Join Calculus* es un cálculo de procesos, como los presentados en las secciones anteriores. Sus nociones primitivas son canales con nombre y procesos que se comunican sobre ellos.

Los canales tienen un lado que recibe y uno que emite. El lado que recibe está asociado a un y sólo un proceso que es activado cada vez que se recibe algo sobre el canal (corresponde a un proceso de la forma $!x(y).P$ en el cálculo π). El lado emisor es el que se representa por el nombre y puede ser transmitido o usado para transmitir otros datos sobre él.

Hay un concepto de localidades anidadas. Una localidad puede migrar a voluntad, junto con todas sus sublocalidades. El cálculo migra automáticamente (por medio de una de las reglas estructurales) una emisión a la localidad de definición del canal correspondiente, en donde es consumida. De esta forma se evitan problemas de sincronización, pues un proceso emisor no necesita coordinarse con otros para transmitir sobre un canal dado.

Este mecanismo también oculta a los procesos la estructura de las localidades. Una comunicación local es indistinguible de una remota.

[3] Por ejemplo, la definición utiliza el concepto de 'multiconjunto'.

El cálculo define la noción de fallo de una localidad. El resto de las localidades pueden detectar este fallo y actuar en consecuencia.

El cálculo se puede codificar dentro de un subconjunto del mismo cálculo que carece de localidades. Con esto se demuestra que, siempre que no hayan fallas, las localidades no tienen significado semántico, sino sólo pragmático (velocidad de ejecución, por ejemplo). A su vez, este fragmento reducido puede codificar al cálculo π, lo que demuestra su universalidad como mecanismo de cómputo.

El *Join Calculus* es la base de JoCaml, un lenguaje para cómputo distribuido [17]. El *Join Calculus* no llena los requisitos para ser un modelo para el cálculo móvil. Entre sus problemas principales se encuentran:

Las localidades se suponen permanentemente conectadas. El cálculo supone que siempre es posible comunicarse entre dos localidades cualesquiera. Un fallo en la comunicación es considerado como una falla permanente de la localidad.

No hay barreras entre ellas. El cálculo supone que el conocimiento de un nombre es permiso suficiente para comunicarse con él. No hay forma de establecer sistemas de permisos.

La comunicación es 'gratuita'. El cálculo implementa la comunicación como una operación atómica. No hay forma de distinguir, desde dentro del modelo, entre una comunicación local y una remota, por lo que no es posible predecir el impacto en el desempeño de una operación dada.

El ruteo es transparente. No hay forma de exponer a un proceso las decisiones sobre la ruta que toma un mensaje entre origen y destino. Distintas aplicaciones tienen necesidades distintas en este sentido. Por ejemplo, se puede sacrificar latencia por ancho de banda.

1.1.3. *Mobile UNITY*

Mobile UNITY es un modelo propuesto por G. Roman, P. McCann y J. Plun en 1997 [33]. El modelo extiende una lógica temporal llamada *UNITY* [14] para dar a cada programa una noción de localización. Además se permite a los programas compartir variables y establecer sincronización siempre que sus posiciones son "cercanas". Se hereda de *UNITY* un modelo de demostración de propiedades de los programas, que es extendido para tomar en cuenta los nuevos mecanismos.

Mobile UNITY está más enfocado a especificar el comportamiento de un agente móvil como entidad aislada. El concepto de posición es completamente abstracto para el modelo. La noción de cercanía se deja también para ser definida por el sistema bajo estudio. El modelo supone que los agentes están todo el tiempo bajo el control de un supervisor que tiene las reglas bajo las cuales existe comunicación o sincronización entre procesos y que controla la ejecución de instrucciones del agente. Dichas reglas son proporcionadas por el diseñador del sistema y son globales a éste.

Puesto que la comunicación es controlada por el sistema, fallos en la misma están ocultos a los programas, por lo que éstos no tienen forma de responder a ellos. Por otro lado, el modelo no tiene ningún soporte para la existencia de fronteras entre localidades. Cualquier soporte para ésto debe ser incluido en las reglas de interacción.

1.1.4. Cálculo de ambientes

El cálculo de ambientes fue introducido por Gordon y Cardelli [12] con el propósito explícito de modelar las características distintivas de un sistema de cómputo móvil.

El concepto principal es el de *ambiente,* que denota un lugar acotado por una frontera en el que se ejecutan procesos de cómputo. El hecho de que un ambiente tenga una frontera bien definida, que permite determinar con precisión qué se mueve cuando un ambiente migra.

Los ambientes están anidados unos en otros, formando una estructura de árbol. La movilidad se expresa como navegación en esta jerarquía. A nivel conceptual los ambientes pueden representar fronteras lógicas, físicas

o administrativas. Por ejemplo un ambiente puede representar una *laptop* que migra de un ambiente *oficina* hacia un ambiente *casa*.

Cada ambiente contiene una colección de procesos y un conjunto de subambientes. El ambiente, cuando se mueve, lo hace como una unidad, llevándose consigo a sus procesos y subambientes. Esta movilidad está bajo el control de los procesos que viven directamente en el ambiente.

Cada ambiente tiene además un nombre, que es usado para controlar el acceso al mismo.

El siguiente capítulo está dedicado a una exposición completa del modelo.

1.2. Lenguajes de programación

En esta sección hacemos una breve descripción de algunos lenguajes de programación que son antecedentes de este trabajo. La lista no pretende ser exhaustiva, sino sólo enumerar los más relevantes.

Ocamm Lenguaje para cómputo paralelo usado por las Transputers [22]. Se basa en CSP, con una sintaxis semejante a la de Pascal [20, 23].

Concurrent ML Un lenguaje para cómputo paralelo. Consta básicamente de la adición de canales con nombre a ML [31].

Actors Uno de los primeros sistemas para cómputo distribuido, desarrollado en el Grupo de Semánticas de Paso de Mensajes del MIT [1], precede por varios años al cálculo π. Desafortunadamente la compañía que lo distribuía parece haber desaparecido por lo que no es posible encontrar referencias públicas del lenguaje. Se basa en el concepto de procesos concurrentes que interactúan por medio de paso de mensajes asíncronos. Es un modelo de computación concebido para analizar y construir sistemas distribuidos de gran escala. El modelo es muy poderoso porque provee mecanismos para el crecimiento dinámico y reconfiguración que usó el enfoqué de sistemas abiertos.

Linda Lenguaje para cómputo distribuido, que se basa en el concepto de sincronización [16]. Procesos secuenciales concurrentes operan mediante el acceso a un multiconjunto compartido de tuplas, que son secuencias finitas de pares tipo valor. El concepto puede ser añadido a cualquier lenguaje secuencial y ha sido probado por lo menos con C, Scheme, Modula-2 y Eiffel.

Jocaml Lenguaje de investigación basado en el *Join Calculus,* implementado como una extensión a Ocaml [17]. No parece tener mucho uso fuera de su laboratorio de origen. Al igual que el cálculo que le sirve de base está orientado al cómputo distribuido.

Obliq Lenguaje orientado a objetos basado en Modula [9, 8, 6, 5] para cómputo distribuido, que no ha sido muy usado fuera de su laboratorio de origen. Diseñado por Luca Cardelli, es la base de muchas ideas que después se incorporaron en el cálculo de ambientes.

Erlang Lenguaje funcional enfocado al cómputo distribuido, desarrollado por Ericson y que ha resultado bastante exitoso para la construcción de aplicaciones de comunicaciones [2, 18].

1.2.1. Compiladores e intérpretes

Existen diversos compiladores e intérpretes destinados al cómputo distribuido. Además de los de los lenguajes mencionados en la sección anterior, vale la pena mencionar a PICT [37], que es una máquina virtual para el cálculo π. Está implementada como un compilador de un lenguaje basado en el cálculo π al lenguaje C y tiene como un objetivo explícito el permitir usar dicho lenguaje como un "lenguaje máquina" para la implementación de otros lenguajes de más alto nivel para el cómputo distribuido. El diseño de nuestra máquina está basado fuertemente en éste.

En el campo del cómputo móvil, existe una implementación del cálculo de ambientes llamada Ambicobs [30]. Diseñada como un *applet* de Java, tiene como propósito principal la visualización de la interacción entre procesos. Cada proceso es representado como una "burbuja", que interactúa con las otras "burbujas" presentes en el entorno, que es completamente gráfico.

Capítulo 2

El cálculo de ambientes

Luca Cardelli y Andrew Gordon diseñaron en 1998-1999 un cálculo que toma en cuenta las características especiales de una WAN que se han mencionado previamente [12]. El componente principal de este cálculo son los ambientes: unidades de movilidad acotadas, con una frontera bien definida.

Cardelli y Gordon postulan que el principal concepto nuevo que se debe tratar al moverse del cómputo distribuido al cómputo móvil es el de una frontera. En las secciones anteriores mencionamos como una de las deficiencias del cálculo π y el *Join Calculus* el hecho de que se supone que cualquier proceso puede comunicarse con cualquier otro que conozca en cualquier momento. En una WAN esto no es cierto. Es posible que la comunicación entre dos procesos se vea interrumpida de forma temporal, ya sea por razones físicas, como la pérdida de un enlace, o virtuales, por la presencia de un *firewall*.

El cálculo de ambientes (CA) modela esto por medio del concepto de *ambientes*. Intuitivamente, un ambiente es un lugar bien delimitado dentro del cual se lleva a cabo el proceso de cómputo. Ejemplos de ambientes serían una computadora, un espacio de direcciones o una red local. Los ambientes están anidados. Cuando un ambiente se mueve, lo hace como un todo, llevando consigo sus procesos y sus subambientes.

Los ambientes representan también piezas de código móvil. Así, un ambiente puede comenzar su vida dentro de una cierta computadora, para después migrar junto con todos sus datos y su estado interno a alguna

otra.

El movimiento de los ambientes está controlado por *capacidades*. Éstas se derivan a partir de los nombres de los ambientes. El nombre de un ambiente es una característica única del mismo y se supone infalsificable.

Dentro de cada ambiente, la comunicación entre procesos se realiza como en cálculos previos. Pero la comunicación remota está estrictamente controlada por el mecanismo de capacidades. Así, diferentes ambientes pueden controlar la comunicación que entra y sale de ellos, simulando ya sea controles administrativos o condiciones físicas.

El alcance de los nombres en un proceso sigue estrictamente reglas estáticas, lo que permite, por ejemplo, la existencia de sistemas de tipos.

Descripción formal

Suponemos la existencia de un conjunto infinito de nombres de ambientes. Denotamos a sus elementos por n, m, etc.

Las expresiones consisten en nombres, capacidades o cadenas de capacidades y son especificadas por la siguiente gramática.

$M, N :=$	
ϵ	nula
$\mid n$	nombre de ambiente
$\mid \text{in}\, n$	entrada al ambiente n
$\mid \text{out}\, n$	salida del ambiente n
$\mid \text{open}\, n$	disolución del ambiente n
$\mid M.N$	prefijo

Un proceso puede ser el proceso nulo, la composición paralela de procesos, la replicación de un proceso, la restricción de un nombre a un proceso, el uso de una capacidad, la creación de un ambiente, la emisión de una

expresión o la recepción de una expresión.

$P :=$	
0	proceso nulo
$\mid P \mid Q$	composición paralela
$\mid !P$	replicación
$\mid (\nu\, n)P$	restricción
$\mid M.P$	ejecución de capacidad
$\mid M[P]$	creación de un ambiente
$\mid \langle M \rangle$	comunicación (envío)
$\mid (n).P$	comunicación (recepción)

Las operaciones de restricción y de recepción acotan el nombre usado en el proceso. Consideramos idénticos a procesos que difieren por la sustitución consistente de nombres acotados.

Como en el cálculo π, presentado en la sección 1.1.1, definimos una relación de congruencia estructural entre procesos, que identifica a procesos semánticamente equivalentes.

La figura 2.2 da las reglas de reducción del cálculo. Decimos que un proceso P *reduce a* otro Q si $P \equiv \longrightarrow^* \equiv Q$.

Una característica común a todos los cálculos descritos hasta ahora, incluyendo el cálculo de ambientes, es que carecen de la propiedad de *confluencia*. Esto quiere decir que, cuando es posible aplicar más de una regla de reducción a una expresión, la elección puede dar lugar a resultados muy diferentes en cada caso, que posiblemente no vuelven a llegar a una forma común.

Tomemos por ejemplo el proceso

$$\text{open}\, n \mid m[\text{in}\, n] \mid n[].$$

En esta expresión podemos aplicar tanto la regla *Open* como la regla *In*. En el primer caso llegamos a la expresión

$$m[\text{in}\, n],$$

$P \mid Q \equiv Q \mid P$	$(ParCom)$
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$(ParAsoc)$
$!P \equiv P \mid !P$	$(ParRepl)$
$(\nu\, m)(\nu\, n)P \equiv (\nu\, n)(\nu\, m)P$	$(ResRes)$
$(\nu\, n)(P \mid Q) \equiv P \mid (\nu\, n)Q \quad \text{si } n \notin fn(Q)$	$(ResPar)$
$(\nu\, n)m[P] \equiv m[(\nu\, n)P] \quad \text{si } n \neq m$	$(ResAmb)$
$P \mid 0 \equiv P$	$(ParCero)$
$(\nu\, m)0 \equiv 0$	$(nuResCero)$
$!0 \equiv 0$	$(ReplCero)$
$\epsilon.P \equiv P$	(ϵ)
$(M.N).P \equiv M.N.P$	$(.)$
$P \equiv Q \Longrightarrow (\nu\, n)P \equiv (\nu\, n)Q$	(Res)
$P \equiv Q \Longrightarrow P \mid R \equiv Q \mid R$	(Par)
$P \equiv Q \Longrightarrow !P \equiv !Q$	$(Repl)$
$P \equiv Q \Longrightarrow n[P] \equiv m[Q]$	(Amb)
$P \equiv Q \Longrightarrow M.P \equiv M.Q$	$(Acción)$
$P \equiv Q \Longrightarrow (n).P \equiv (n).Q$	$(Entrada)$

Figura 2.1: Congruencia estructural del cálculo de ambientes

que ya no se puede reducir más, mientras que en el segundo llegamos a

$$\text{open}\, n \mid n[m[]],$$

que a su vez se reduce a

$$m[].$$

Claramente ambas expresiones no son equivalentes.

Limitaciones

El CA es una herramienta teórica para el cómputo móvil ampliamente aceptada por la comunidad. Está explícitamente diseñado para atacar los problemas especiales de este entorno.

Sin embargo, no existe una validación práctica de su utilidad. Para tener ésta sería necesario usarlo para resolver problemas específicos en el contexto del cómputo móvil, usándolo como lenguaje para describir la solución.

$$
\begin{array}{ll}
m[\text{in } n.P \mid Q] \mid n[R] \longrightarrow n[m[P \mid Q] \mid R] & (In) \\
n[m[\text{out } n.P \mid Q] \mid R] \longrightarrow n[R] \mid m[P \mid Q] & (Out) \\
m[\text{open } n.P \mid n[R \mid S] \mid Q] \longrightarrow m[P \mid R \mid S \mid Q] & (Open) \\
m[\langle M \rangle \mid (n).P \mid Q] \longrightarrow m[P\{n := M\} \mid Q] & (Comm) \\
P \longrightarrow Q \Longrightarrow n[P] \longrightarrow n[Q] & (RedAmb) \\
P \longrightarrow Q \Longrightarrow (\nu\, n)P \longrightarrow (\nu\, n)Q & (RedRes) \\
P \longrightarrow Q \Longrightarrow P \mid R \longrightarrow Q \mid R & (RedPar)
\end{array}
$$

Figura 2.2: Semántica operacional del cálculo de ambientes

El CA está limitado en este respecto. Su origen como herramienta teórica dicta características que lo hacen poco apto para expresar soluciones directas a problemas específicos. Entre los más notorios tenemos que la sintaxis es sumamente concisa, lo que dificulta su lectura cuando se excede una cierta longitud.

Además, es conceptualmente muy económico. El cálculo no incluye ninguna operación que en principio puede ser definida a partir de las primitivas incluidas. Esto facilita el análisis algebraico, pero complica la construcción de soluciones complejas.

Variaciones

A partir del trabajo de Gordon y Cardelli se han publicado en la literatura algunas variaciones sobre el cálculo original. Estas variaciones tienen como propósito adecuar el cálculo al estudio de algún fenómeno en particular. Entre las más comunes se encuentran:

Sistemas de tipos Éstos han sido usados para distinguir la clase de comunicaciones que ocurren dentro de un ambiente. A partir de esto es posible proporcionar garantías acerca del comportamiento del mismo. Por ejemplo, es posible aseverar que un ambiente es inamovible, o que nunca disolverá a un subambiente [11, 38, 13, 10].

Distintas capacidades Algunos artículos han explorado el variar las capacidades disponibles y el efecto que esto tiene sobre la expresividad del

cálculo resultante. Una variación interesante es la introducción de *cocapacidades*, que representan el permiso interno de un ambiente para que otro haga uso de una capacidad sobre él [25].

Canales Una variación interesante fue introducida por Pericas [29] en su tesis doctoral. Consiste en introducir en cada ambiente canales con nombre sobre los que se lleva a cabo la comunicación. Esto permite que se lleven a cabo varias conversaciones independientes dentro de un ambiente sin tener que ejecutar protocolos de sincronización.

Un resultado interesante de estos trabajos es que la capacidad expresiva del cálculo en general no se ve afectada. Esto nos permite elegir las extensiones que más se ajusten a nuestras necesidades sin problemas.

Capítulo 3

Máquina abstracta

En este capítulo presentamos una máquina virtual para ejecutar el cálculo de ambientes. El diseño de esta máquina está fuertemente basado en el de Pict [37].

Éste diseño no es una implementación distribuida, aunque se trató de dejarlo lista para que en una versión posterior se añada esta característica.

La máquina ha sido implementada en el lenguaje *Haskell* [24]. Dicha implementación se presenta en el capítulo 4.

3.1. Lenguaje fuente

La máquina ejecuta procesos en un lenguaje que es una ligera modificación del cálculo de ambientes. La primera diferencia es que se añade una clase de procesos simples S, para hacer explícitas las reglas de precedencia en la sintaxis.

Un proceso P es un proceso simple S o una composición paralela de procesos simples.

$$\begin{array}{rl} P := & \quad | \\ & S_1 | \, S_2 | \ldots | \, S_n \end{array}$$

Un proceso simple es una de las formas primitivas del cálculo de ambientes, o un proceso encerrado en paréntesis.

$$
\begin{array}{rl}
S := & P) \quad | \\
& 0 \quad | \\
& M[P] \quad | \\
& (\nu n)S \quad | \\
& \langle M \rangle \quad | \\
& (x).S \quad | \\
& M.S \quad | \\
& !M.S \quad | \\
& !(x).S
\end{array}
$$

Las expresiones son como en el cálculo de ambientes.

$$
\begin{array}{rl}
M := & \quad | \\
& n \quad | \\
& \text{in } n \quad | \\
& \text{out } n \quad | \\
& \text{open } n \quad | \\
& M.M'
\end{array}
$$

La segunda diferencia es que se elimina la construcción $!P$. En su lugar tenemos $!M.S$ y $!(x).S$. Esto se hace por razones de eficiencia en la implementación. En el cálculo original la semántica de $!P$ está dada por la regla estructural $!P \equiv !P|P$. Esta regla no nos da información acerca del momento adecuado para crear una nueva copia del proceso P. En este lenguaje sustituimos esta regla estructural por las siguientes reducciones, que en el cálculo original son teoremas:

$$!(x).P|\langle M\rangle \longrightarrow !(x).P|P\{x \to M\} \tag{3.1}$$

$$!\,\text{open } n.P|\,n[Q] \longrightarrow !\,\text{open } n.P|\,P|\,Q \tag{3.2}$$

$$n[!\,\text{in } m.P|\,Q]|\,m[R] \longrightarrow m[n[!\,\text{in } m.P|\,P|\,Q]|\,R] \tag{3.3}$$

$$n[m[!\,\text{out } n.P|\,Q]|\,R] \longrightarrow n[R]|\,m[!\,\text{out } n.P|\,P|\,Q] \tag{3.4}$$

Puesto que $!P$ se puede codificar como $!(x).(\langle x\rangle|\,P)|\,\langle x\rangle$ no perdemos expresividad en el lenguaje al hacer esta modificación, aunque debe notarse

que esta codificación sería muy ineficiente en una implementación, puesto que crearía un número ilimitado de copias de P.

Los nombres se toman de un conjunto infinito numerable de nombres que denotamos $\mathcal{N}$.

Denotamos al conjunto de los procesos por $\mathcal{P}$. Nombres y procesos son fundamentales para la definición del estado interno de la máquina.

3.2. Estado interno

Consideremos el conjunto $\mathcal{C} = \{\text{in}\, n, \text{out}\, n, \text{open}\, n | n \in \mathcal{N}\} \cup \{\cdot\}$ de capacidades y "comunicación". Un *heap* es una función parcial $\mathcal{C} \mapsto S^*$. La intención del *heap* es mantener listas de espera de procesos que no tienen las condiciones necesarias para ejecutarse. Sea H un *heap*. En el caso de una capacidad c, $H(c)$ es una lista, posiblemente vacía, de procesos de la forma c.P o !c.P. En el caso de comunicaciones, $c = \cdot$ y $H(c)$ es una lista de procesos de la forma $(x).P$ o $\langle M \rangle$. Denotamos por $\mathcal{H}$ al conjunto de heaps.

Si H es un heap, $H\{c := X\}$ es el heap dado por

$$H\{c := X\}(y) = \begin{cases} H(y) & \text{si } y \neq c, \\ X & \text{si } y = c. \end{cases}$$

Sea Top un nombre distinguido, que no pertenece a $\mathcal{N}$ y que, por lo tanto, no puede aparecer en la sintaxis del lenguaje.

Definimos el conjunto de los ambientes $\mathcal{AMB} = (\mathcal{N} \cup \{\text{Top}\}) \times \mathcal{P}(\mathcal{P}) \times \mathcal{H} \times \mathcal{P}(\mathcal{AMB})$. Un ambiente $A \in \mathcal{AMB}$ consta de un nombre, un multiconjunto de procesos, un *heap* y un conjunto de hijos. Denotamos a sus componentes con una notación estilo POO: $A = (A.\text{nombre}, A.\text{procs}, A.\text{heap}, A.\text{hijos})$.

Si $A, B \in \mathcal{AMB}$ definimos $A \oplus B = (A.\text{nombre}, A.\text{procs}, A.\text{heap}, A.\text{hijos} \cup \{B\})$ como el resultado de añadir a B como hijo de A.

Similarmente $A \ominus B = (A.\text{nombre}, A.\text{procs}, A.\text{heap}, A.\text{hijos}/\{B\})$ representa al ambiente A menos el ambiente B.

Si $A \in \mathcal{AMB}$ y $P \in \mathcal{P}$ definimos $A \oplus P = (A.\text{nombre}, A.\text{procs} \cup \{P\}, A.\text{heap}, A.\text{hijos})$ como el resultado de añadir a P al conjunto de procesos de A.

Igualmente $A \ominus P = (A.\mathit{nombre}, A.\mathit{procs}/\{P\}, A.\mathit{heap}, A.\mathit{hijos})$ es el resultado de eliminar (una copia de) P del conjunto de procesos de A. La ambigüedad entre estas dos notaciones y las anteriores se resuelve por el contexto. Nótese que ninguna de estas operaciones es asociativa o conmutativa.

La notación $A\{X \to X'\}$ es una abreviación para $A \ominus X \oplus X'$, en donde X y X′ son ambos procesos o ambos ambientes.

Decimos que un conjunto $\boldsymbol{\mathcal{A}} \subset \mathcal{AMB}$ de ambientes es elegible si

1. $\boldsymbol{\mathcal{A}}$ es finito y no vacío.
2. Hay un y sólo un ambiente $T \in \boldsymbol{\mathcal{A}}$ tal que $T.\mathit{nombre} = \mathrm{Top}$.
3. Si $A \in \boldsymbol{\mathcal{A}}$, $\forall B \in A.\mathit{hijos}$, $B \in \boldsymbol{\mathcal{A}}$.
4. $\boldsymbol{\mathcal{A}}$, con la relación es-hijo, es un árbol, donde la relación está definida por B es-hijo $A \iff B \in A.\mathit{hijos}$.

Denotamos por $\hat{\boldsymbol{\mathcal{A}}} \subset \mathcal{P}(\mathcal{AMB})$ el conjunto de todos los conjuntos de ambientes elegibles.

Definimos $\mathcal{R} = (\mathcal{P} \times \mathcal{AMB})^*$ como el conjunto de colas de ejecución. Una cola de ejecución $R \in \mathcal{R}$ es una lista, posiblemente vacía, de pares (P, A), en donde $P \in A.\mathit{procs}$.

$\mathcal{S} = \mathcal{R} \times \hat{\boldsymbol{\mathcal{A}}}$ es el conjunto de estados de la máquina virtual. Un estado Σ es un par de una cola de ejecución R y un conjunto elegible de ambientes $\boldsymbol{\mathcal{A}} \in \hat{\boldsymbol{\mathcal{A}}}$ tales que si $(P, A) \in R$, $A \in \boldsymbol{\mathcal{A}}$.

Dado un proceso P, el estado inicial de la máquina que lo ejecuta es $([(P, T)], \{T\})$, en donde $T = (\mathrm{Top}, \{P\}, \emptyset, \emptyset)$.

3.3. Reglas de transición

Presentamos en esta sección las reglas de transición de la máquina virtual. Hay dos clases de reglas. Unas implementan las reglas de la semántica operacional del cálculo de ambientes. Las otras son transiciones internas de la máquina. En el caso de las primeras se muestra entre paréntesis la regla del cálculo de ambientes que implementan.

La transición a realizar es escogida según el primer proceso en la cola. Un proceso 0 simplemente se elimina de la cola y de su ambiente.

$$\frac{(P \mid 0 \equiv P) \qquad P = 0, A' = A \ominus P}{((P, A) : R, \mathcal{A}) \longrightarrow (R, \mathcal{A}\{A \to A'\})} \text{(Null)}$$

Puesto que la máquina está diseñada para ejecutarse en una máquina secuencial no puede implementar directamente la composición paralela. Siguiendo el ejemplo de Pict simulamos esta última por medio de la ejecución intercalada de los componentes, por lo que una composición paralela es sustituida por sus componentes en orden.

$$\frac{P = P_1 \mid P_2 \mid \ldots \mid P_n, A' = A\{P \to \{P_1, P_2, \ldots, P_n\}\}}{((P, A) : R, \mathcal{A}) \longrightarrow (\ (P_1, A) : (P_2, A) : \cdots : (P_n, A) : R, \ \mathcal{A}\{A \to A'\})} \text{(Par)}$$

La restricción de nombres es implementada sustituyendo el nombre dado por un nuevo nombre globalmente único.

$$\frac{n \text{ único}, P = (\nu n)Q, P' = Q\{c \to n\}, A' = A\{P \to P'\}}{((P, A) : R, \mathcal{A}) \longrightarrow ((P', A) : R, \mathcal{A}\{A \to A'\})} \text{(Rest)}$$

La creación de un nuevo ambiente tiene tres casos. El primero es cuando hay un $\text{open}\, n.P$ en el Heap del ambiente correspondiente. Si es así, lo ejecutamos directamente sin crear el ambiente.

$$\frac{\begin{array}{c}(\text{open}\, n.S \mid n[Q] \longrightarrow S \mid Q) \\ P = n[Q], A.heap(\text{open}\, n) = \text{open}\, n.S : \vec{S}, \\ H' = A.heap\{\text{open}\, n := \vec{S}\}, A' = A\{P \to Q\}\{A.heap \to H'\} \oplus S\end{array}}{((P, A) : R, \mathcal{A}) \longrightarrow ((Q, A) : R : (S, A), \mathcal{A}\{A \to A'\})} \text{(OpenAmb)}$$

El caso de un $!\,\text{open}\, n.P$ es similar, salvo que dejamos una copia en el Heap.

$$\frac{\begin{array}{c}(!\,\text{open}\, n.S \mid n[Q] \longrightarrow !\,\text{open}\, n.S \mid S \mid Q) \\ P = n[Q], A.heap(\text{open}\, n) = !\,\text{open}\, n.S : \vec{S}, \\ H' = A.heap\{\text{open}\, n := \vec{S} : !\,\text{open}\, n.S\}, A' = A\{P \to Q\}\{A.heap \to H'\} \oplus S\end{array}}{((P, A) : R, \mathcal{A}) \longrightarrow ((Q, A) : R : (S, A), \mathcal{A}\{A \to A'\})} \text{(BOpenAmb)}$$

Por otro lado, si el Heap está vacío, procedemos a crear el ambiente y a ejecutar todos los $\text{in}\, n.P$ y $!\,\text{in}\, n.P$ que estaban esperando entrar a un ambiente de este nombre.

$$\frac{\begin{array}{l}(n[Q] \mid B_i[\text{in}\, n.S_i \mid \vec{S_i}] \mid D_j[!\,\text{in}\, n.R_j \mid \vec{R_j}] \longrightarrow n[Q \mid B_i[\vec{S_i}] \mid D_j[!\,\text{in}\, n.R_j \mid \vec{R_j}]]) \\ P = n[Q], \\ \forall B_i \in A, B_i.heap(\text{in}\, n) = \text{in}\, n.S_i : \vec{S_i} \\ \forall D_j \in A, D_j.heap(\text{in}\, n) = !\,\text{in}\, n.R_j : \vec{R_j} \\ C \notin \boldsymbol{\mathcal{A}}, C.name = n \\ H'_{B_i} = B_i.heap\{\text{in}\, n := \vec{S_i}\} \\ H'_{D_j} = D_j.heap\{\text{in}\, n := \vec{R_j} : !\,\text{in}\, n.R_j\} \\ B'_i = (B_i\{B_i.heap \to H'_{B_i}\}) \oplus S_i \\ D'_j = (D_j\{D_j.heap \to H'_{D_j}\}) \oplus R_i \\ C' = C \oplus Q \oplus B'_1 \oplus \cdots \oplus B'_l \oplus D'_1 \oplus \cdots \oplus D_m, \\ A' = (A \ominus P \ominus B_1 \ominus \cdots \ominus B_l \ominus D_1 \ominus \cdots \ominus D_m) \oplus C'\end{array}}{\begin{array}{c}((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (\ (Q, A) : R : (S_1, B'_1) : \cdots : (S_n, B'_n) : (R_1, D'_i) : \cdots : (R_m, D'_m), \\ \boldsymbol{\mathcal{A}}\{A \to A'\}\{B_i \to B'_i\}\{D_j \to D'_j\} \oplus C')\end{array}} \quad \text{(NewAmb)}$$

Tres casos para una escritura. El primero es que el Heap esté vacío o contenga otros escritores.

$$\frac{\begin{array}{c}P = \langle M \rangle, A.heap(\cdot) = \vec{W} \\ H' = A.heap\{\cdot := \vec{W} : P\} \\ A' = A\{A.heap \to H'\} \ominus P\end{array}}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (R, \boldsymbol{\mathcal{A}}\{A \to A'\})} \quad \text{(NoReaders)}$$

El segundo es que haya un lector

$$\frac{\begin{array}{c}(\langle M \rangle \mid (x).Q \longrightarrow Q\{x \to M\}) \\ P = \langle M \rangle, A.heap(\cdot) = (x).Q : \vec{R} \\ P' = Q\{x \to M\} \\ H' = A.heap\{\cdot := \vec{R}\} \\ A' = A\{A.heap \to H'\}\{P \to P'\}\end{array}}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (R : (P', A), \boldsymbol{\mathcal{A}}\{A \to A'\})} \quad \text{(Reader)}$$

y el tercero es un lector replicado.

$$\frac{\begin{array}{c}(\langle M\rangle|\ !(x).Q \longrightarrow Q\{x \to M\}|\ !(x).Q)\\ P = \langle M\rangle, A.heap(\cdot) = !(x).Q : \vec{R}\\ P' = Q\{x \to M\}\\ H' = A.heap\{\cdot := \vec{R} : !(x).Q\}\\ A' = A\{A.heap \to H'\}\{P \to P'\}\end{array}}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (R : (P', A), \boldsymbol{\mathcal{A}}\{A \to A'\})} \quad \text{(BReader)}$$

La lectura sólo tiene dos casos. El primero es si el Heap no tiene escritores

$$\frac{P = (x).Q, A.heap(\cdot) = \vec{R}, H' = A.heap\{\cdot := \vec{R} : P\}, A' = A\{A.heap \to H'\} \ominus P}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (R, \boldsymbol{\mathcal{A}}\{A \to A'\}} \quad \text{(NoWriters)}$$

mientras que el segundo es cuando sí.

$$\frac{\begin{array}{c}(\langle M\rangle|\ (x).Q \longrightarrow Q\{x \to M\})\\ P = (x).Q, A.heap(\cdot) = \langle M\rangle : \vec{W}\\ P' = Q\{x \to M\}\\ H' = A.heap\{\cdot := \vec{W}\}\\ A' = A\{A.heap \to H'\}\{P \to P'\}\end{array}}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow ((P', A) : R, \boldsymbol{\mathcal{A}}\{A \to A'\})} \quad \text{(Writer)}$$

La lectura replicada es muy similar, salvo que en el segundo caso no se quita al proceso de la cola de ejecución, para que tenga oportunidad de tomar otro escritor del Heap.

$$\frac{\begin{array}{c}(\langle M\rangle|\ !(x).Q \longrightarrow Q\{x \to M\}|\ !(x).Q)\\ P = !(x).Q, A.heap(\cdot) = \vec{R}, H' = A.heap\{\cdot := \vec{R} : P\}, A' = A\{A.heap \to H'\} \ominus P\end{array}}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (R, \boldsymbol{\mathcal{A}}\{A \to A'\}} \quad \text{(BNoWriters)}$$

$$\frac{\begin{array}{c}P = !(x).Q, A.heap(\cdot) = \langle M\rangle : \vec{W}\\ P' = Q\{x \to M\}\\ H' = A.heap\{\cdot := \vec{W}\}\\ A' = A\{A.heap \to H'\}\{P \to P'\}\end{array}}{((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow ((P, A) : R : (P, A), \boldsymbol{\mathcal{A}}\{A \to A'\})} \quad \text{(BWriter)}$$

Para la ejecución de capacidades la condición a revisar es la existencia de un ambiente con el nombre adecuado. Si éste existe se ejecuta el proceso, si no, se le sube al Heap adecuado, de donde será despertado cuando las condiciones cambien.

Para $\mathrm{in}\, n.Q$ debemos buscar un ambiente hermano

$$\frac{\begin{array}{l}(A[\mathrm{in}\, n.Q \mid Q'] \mid n[R] \longrightarrow n[R \mid A[Q \mid Q']]) \\ P = \mathrm{in}\, n.Q, B \in A\uparrow, B.\mathrm{nombre} = n \\ A.\mathrm{heap}(\mathrm{out}\, n) = S : \vec{S} \\ \forall C_i \in B, A.\mathrm{heap}(\mathrm{in}\, C_i.\mathrm{nombre}) = T_i : \vec{T}_i \\ C_i.\mathrm{heap}(\mathrm{in}\, n) = U_i : \vec{U}_i \\ B.\mathrm{heap}(\mathrm{open}\, n) = V : \vec{V} \\ H'_A = A.\mathrm{heap}\{\mathrm{out}\, n := \vec{S}, \mathrm{in}(C_i.\mathrm{nombre}) = \vec{T}_i\} \\ H'_{C_i} = C_i.\mathrm{heap}\{\mathrm{in}\, n := \vec{U}_i\} \\ H'_B = B.\mathrm{heap}\{\mathrm{open}\, n := \vec{V}\} \\ A\uparrow' = A\uparrow \ominus A \\ B' = B\{B.\mathrm{heap} \to H'_B\} \oplus V \oplus A \\ C'_i = C_i\{C_i.\mathrm{heap} \to H'_{C_i}\} \oplus U_i \\ A' = A\{A.\mathrm{heap} \to H'_A, P \to Q\} \oplus S \oplus T_1 \oplus \cdots \oplus T_l\end{array}}{\begin{array}{l}((P, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow ((Q, A) : R : (S, A) : (T_i, A) : (U_i, C_i) : (V, B), \\ \qquad\qquad \boldsymbol{\mathcal{A}}\{A\uparrow \to A\uparrow', B \to B', C_i \to C'_i, A \to A'\})\end{array}} \quad (\mathrm{In})$$

al igual que para $!\,\mathrm{in}\, n.Q$, salvo que en este caso no quitamos al proceso

de la cola de ejecución.

$$\frac{\begin{array}{l}(A[!\,\mathrm{in}\,n.Q \mid Q'] \mid n[R] \longrightarrow n[R \mid A[!\,\mathrm{in}\,n.Q \mid Q \mid Q']]) \\ P = !\,\mathrm{in}\,n.Q, B \in A\uparrow, B.nombre = n \\ A.heap(\mathrm{out}\,n) = S : \vec{S} \\ \forall C_i \in B, A.heap(\mathrm{in}\,C_i.nombre) = T_i : \vec{T_i} \\ C_i.heap(\mathrm{in}\,n) = U_i : \vec{U_i} \\ B.heap(\mathrm{open}\,n) = V : \vec{V} \\ H'_A = A.heap\{\mathrm{out}\,n := \vec{S}, \mathrm{in}(C_i.nombre) = \vec{T_i}\} \\ H'_{C_i} = C_i.heap\{\mathrm{in}\,n := \vec{U_i}\} \\ H'_B = B.heap\{\mathrm{open}\,n := \vec{V}\} \\ A\uparrow' = A\uparrow \ominus A \\ B' = B\{B.heap \to H'_B\} \oplus V \oplus A \\ C'_i = C_i\{C_i.heap \to H'_{C_i}\} \oplus U_i \\ A' = A\{A.heap \to H'_A\} \oplus Q \oplus S \oplus T_1 \oplus \cdots \oplus T_l\end{array}}{\begin{array}{l}((P,A) : R, \mathcal{A}) \longrightarrow ((P,A) : R : (Q,A) : (S,A) : (T_i,A) : (U_i,C_i) : (V,B), \\ \qquad \mathcal{A}\{A\uparrow \to A\uparrow', B \to B', C_i \to C'_i, A \to A'\})\end{array}} \quad \text{(BIn)}$$

En el caso de out n.Q debemos revisar el nombre del padre.

$$\frac{\begin{array}{l}(n[A[\mathrm{out}\,n.Q \mid Q'] \mid R] \longrightarrow n[R] \mid A[Q \mid Q']) \\ P = \mathrm{out}\,n.Q \\ A\uparrow.nombre = n \\ \forall B_i \in A\uparrow\uparrow, A.heap(\mathrm{in}\,B_i.nombre) = S_i : \vec{S_i} \\ A\uparrow\uparrow.heap(\mathrm{open}\,n) = T : \vec{T} \\ A.heap(\mathrm{out}\,A\uparrow\uparrow.nombre) = U : \vec{U} \\ H'_{A\uparrow\uparrow} = A\uparrow\uparrow.heap\{\mathrm{open}\,n := \vec{T}\} \\ H'_A = A.heap\{\mathrm{in}\,B_i.nombre := \vec{S_i}, \mathrm{out}\,A\uparrow\uparrow.nombre := \vec{U}\} \\ A' = A\{A.heap \to H'_A, P \to Q\} \oplus U \oplus S_i \\ A\uparrow' = A\uparrow \ominus A \\ A\uparrow\uparrow' = A\uparrow\uparrow\{A\uparrow\uparrow.heap \to H'_{A\uparrow\uparrow}\} \oplus A' \oplus T\end{array}}{\begin{array}{l}((P,A) : R, \mathcal{A}) \longrightarrow ((Q,A) : R : (S_i,B_i) : (T_i,A) : (U_i,A), \\ \qquad \mathcal{A}\{A\uparrow\uparrow \to A\uparrow\uparrow', A\uparrow \to A\uparrow', A \to A'\})\end{array}} \quad \text{(Out)}$$

$$\frac{\begin{array}{l}(n[A[!\,\mathrm{out}\,n.Q \mid Q'] \mid R] \longrightarrow n[R] \mid A[!\,\mathrm{out}\,n.Q \mid Q \mid Q']) \\ P = !\,\mathrm{out}\,n.Q \\ A\uparrow.nombre = n \\ \forall B_i \in A\uparrow\uparrow, A.heap(\mathrm{in}\,B_i.nombre) = S_i : \vec{S_i} \\ A\uparrow\uparrow.heap(\mathrm{open}\,n) = T : \vec{T} \\ A.heap(\mathrm{out}\,A\uparrow\uparrow.nombre) = U : \vec{U} \\ H'_{A\uparrow\uparrow} = A\uparrow\uparrow.heap\{\mathrm{open}\,n := \vec{T}\} \\ H'_A = A.heap\{\mathrm{in}\,B_i.nombre := \vec{S_i}, \mathrm{out}\,A\uparrow\uparrow.nombre := \vec{U}\} \\ A' = A\{A.heap \to H'_A\} \oplus Q \oplus U \oplus S_i \\ A\uparrow' = A\uparrow \ominus A \\ A\uparrow\uparrow' = A\uparrow\uparrow\{A\uparrow\uparrow.heap \to H'_{A\uparrow\uparrow}\} \oplus A' \oplus T\end{array}}{\begin{array}{l}((P,A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow \ ((P,A) : R : (Q,A) : (S_i,B_i) : (T_i,A) : (U_i,A), \\ \qquad\qquad \boldsymbol{\mathcal{A}}\{A\uparrow\uparrow \to A\uparrow\uparrow', A\uparrow \to A\uparrow', A \to A'\})\end{array}} \quad (\mathrm{BOut})$$

Mientras que para $\mathrm{open}\,n.Q$ se busca un hijo.

$$\frac{\begin{array}{l}(A[\mathrm{open}\,n.Q \mid Q' \mid n[S_i]] \longrightarrow A[Q \mid Q' \mid S_i]) \\ P = \mathrm{open}\,n.Q, \\ B \in A, B.nombre = n, B.procs = S_i, C_i \in B \\ A' = A\{P \to Q\} \ominus B \oplus C_i \oplus S_i\}\end{array}}{((\mathrm{open}\,n.Q, A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow \ ((P,A) : R, \boldsymbol{\mathcal{A}}\{A \to A'\} \ominus B)} \quad (\mathrm{Open})$$

$$\frac{\begin{array}{l}(A[!\,\mathrm{open}\,n.Q \mid Q' \mid n[S_i]] \longrightarrow A[!\,\mathrm{open}\,n.Q \mid Q \mid Q' \mid S_i]) \\ P = !\,\mathrm{open}\,n.Q, \\ B \in A, B.nombre = n, B.procs = S_i, C_i \in B \\ A' = A \oplus Q \ominus B \oplus C_i \oplus S_i\}\end{array}}{((P,A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow \ ((P,A) : R : (Q,A), \boldsymbol{\mathcal{A}}\{A \to A'\} \ominus B)} \quad (\mathrm{BOpen})$$

Las reglas para subir al Heap son genéricas

$$\frac{\begin{array}{l}P = capn.Q, A.heap(capn) = \vec{P} \\ H' = A.heap\{capn := \vec{P} : P\}, A' = A\{A.heap \to H'\} \ominus P\end{array}}{((P,A) : R, \boldsymbol{\mathcal{A}}) \longrightarrow (R, \boldsymbol{\mathcal{A}}\{A \to A'\})} \quad (\mathrm{CapAHeap})$$

$$\frac{P = !capn.Q, A.heap(capn) = \vec{P} \quad H' = A.heap\{capn := \vec{P} : P\}, A' = A\{A.heap \to H'\} \ominus P}{((P, A) : R, \mathcal{A}) \longrightarrow (R, \mathcal{A}\{A \to A'\})} \text{ (BCapAHeap)}$$

en donde $cap \in \{in, out, open\}$.

3.4. Validez

Para demostrar la validez de nuestra máquina comenzamos dando una traducción recursiva $T : \Sigma \mapsto \mathcal{P}$ de estados de la máquina al cálculo de ambientes.

Dado un ambiente $A = (n, (P_1, P_2, \ldots, P_n), H, a)$ su traducción es

$$T[\![(n, (P_1, P_2, \ldots, P_n), H, a)]\!] = n[P_1 | \ldots | P_n | T(H) | T(a).$$

Una lista de procesos se traduce a los mismos procesos en paralelo.

$$T(P_1 : P_2 \cdots : P_n) = P_1 | \ldots | P_n.$$

La traducción de un heap H es la concatenación en paralelo de la traducción de cada una de sus entradas.

$$T(H) = |_{c \in Dom(H)} T(H(c)).$$

La traducción de un conjunto $a = \{A_1, \ldots, A_n\}$ de ambientes es la concatenación en paralelo de la traducción de cada uno.

$$T(a) = T(A_1) | \ldots | T(A_n).$$

Y finalmente, la traducción de un estado $\Sigma = (R, \mathcal{A})$ es la traducción del ambiente Top.

$$T(\Sigma) = T(Top), \text{ para } Top \in \mathcal{A}.$$

Enunciaremos y demostraremos un teorema de validez, que significa que todas las transiciones de la máquina corresponden a transiciones válidas del cálculo.

Teorema 1 (Validez). *Dados dos estados Σ y Σ' de la máquina virtual, tales que $\Sigma \longrightarrow \Sigma'$, tenemos que $T(\Sigma) \equiv\!\!\longrightarrow^* T(\Sigma')$.*

Demostración. La demostración es por casos, sobre cada una de las reglas de la máquina. Para cada una de ellas traducimos el estado inicial y el final, y vemos que el segundo se obtiene del primero por medio de operaciones del cálculo de ambientes.

- (Null) Supongamos que $\Sigma \xrightarrow{\text{Null}} \Sigma'$. Entonces sabemos que $T(\Sigma) = C(P \mid 0)$ y $T(\Sigma') = C(P)$, para algún contexto C y un proceso P. Puesto que $P \equiv P \mid 0$, tenemos que $T(\Sigma) \equiv T(\Sigma')$.

- (Par) Se sigue de que $T(P_1 \mid \ldots \mid P_n) = T(P_1, \ldots, P_n)$.

- (Rest) Tenemos un proceso $P = (\nu n)P'$. La regla lo convierte en $P'\{n \to c\}$. Pero estos dos procesos son equivalentes en el cálculo de ambientes, usando conversión α y extrusión, y el hecho de que c es un nombre fresco.

- (OpenAmb) Por hipótesis de la regla tenemos un ambiente A tal que $T(A) = n[Q] \mid \ldots \mid \text{open}\, n.S \mid S_i$, mientras que después de aplicar la regla tenemos que $T(A') = Q \mid S \mid \ldots \mid S_i$, lo que significa que $T(A) \longrightarrow T(A')$ y por tanto $T(\Sigma) \longrightarrow T(\Sigma')$.

- (BOpenAmb) Análogamente al caso anterior

  $$T(A) = n[Q] \mid \ldots \mid !\,\text{open}\, n.S \mid S_i \equiv n[Q] \mid \text{open}\, n.S \mid \ldots \mid !\,\text{open}\, n.S \mid S_i,$$

  mientras que $T(A') = Q \mid S \mid \ldots \mid !\,\text{open}\, n.S \mid S_i$, de donde tenemos que $T(\Sigma) \equiv\!\!\longrightarrow T(\Sigma')$.

- (NewAmb) La traducción de la hipótesis es

  $$T(A) = m[n[Q] \mid P \mid l_1[T_1 \mid \text{in}\, n.S_1 \mid \vec{S_1}] \ldots k_1[U_1 \mid !\,\text{in}\, n.R_1 \mid \vec{R_1}] \ldots]$$

  mientras que la traducción del consecuente es

  $$T(A') = m[P \mid n[Q \mid l_1[T_1 \mid S_1 \mid \vec{S_1}] \ldots k_1[U_1 \mid R_1 \mid \vec{R_1} \mid !\,\text{in}\, n.R_1] \ldots]]$$

  que se obtiene de la hipótesis por aplicación repetida de la regla para in en el cálculo de ambientes.

- (Reader) $T(A) =< M > | S | (x).Q | \vec{Q}$, mientras que $T(A') = S | Q\{x \to M\} | \vec{Q}$, que se obtiene del anterior por la aplicación directa de la regla de comunicación.
- (BReader) $T(A) =< M > | S | !(x).Q | \vec{Q} \equiv < M > | S | (x).Q | !(x).Q | \vec{Q}$, mientras que $T(A') = S | Q\{x \to M\} | !(x).Q | \vec{Q}$, que se obtiene del anterior por la aplicación directa de la regla de comunicación.
- (Writer) $T(A) = (x).Q | S | < M > | \vec{W}$, mientras que $T(A') = S | Q\{x \to M\} | \vec{W}$, que se obtiene del anterior por la aplicación directa de la regla de comunicación.
- (BWriter) $T(A) = !(x).Q | S | < M > | \vec{W} \equiv (x).Q | !(x).Q | S | < M > | \vec{W}$, mientras que $T(A') = !(x).Q | S | Q\{x \to M\} | \vec{W}$, que se obtiene del anterior por la aplicación directa de la regla de comunicación.
- (In)

  $$T(\Sigma) = l[a[in n.Q | Q' | S | \vec{S} | T_1 | \vec{T}_1 \ldots] | \\ n[Q'' | V | \vec{V} | c_1[Q_1''' | U_1 | \vec{U}_1] \ldots] | Q^{(i v)}]$$

  mientras que

  $$T(\Sigma') = l[n[V | Q'' | \vec{V} | a[S | T_1 | \ldots | Q | \vec{S} | \vec{T}_1 \ldots] | \\ c_1[U_1 | Q_1''' | \vec{U}_1 \ldots] \ldots] | Q^{(i v)}]$$

  que se obtiene del anterior aplicando la regla in del cálculo de ambientes y reordenando componentes paralelos.
- (BIn)

  $$T(\Sigma) = l[a[!in n.Q | Q' | S | \vec{S} | T_1 | \vec{T}_1 \ldots] | \\ n[Q'' | V | \vec{V} | c_1[Q_1''' | U_1 | \vec{U}_1] \ldots] | Q^{(i v)}]$$

  mientras que

  $$T(\Sigma') = l[n[V | Q'' | \vec{V} | a[!in n.Q | S | T_1 | \ldots | Q | \vec{S} | \vec{T}_1 \ldots] | \\ c_1[U_1 | Q_1''' | \vec{U}_1 \ldots] \ldots] | Q^{(i v)}]$$

obtenido, en el CA, mediante una aplicación de $!P \equiv P | !P$ y una de la regla in.

- (Out)

$$T(\Sigma) = l[n[a[\text{out } n.Q | Q_a | U | \vec{U} | S_1 | \vec{S_1} \ldots] | Q_n] | b_1[Q_{b_1}] | \ldots T | \vec{T}]$$

$$T(\Sigma') = l[n[Q_n] | b_1[Q_{b_1}] | \ldots a[Q | U | \vec{U} | S_1 | \vec{S_1} | \ldots] | T | \vec{T}]$$

- (BOut)

$$T(\Sigma) = l[n[a[! \text{ out } n.Q | Q_a | U | \vec{U} | S_1 | \vec{S_1} \ldots] | Q_n] | b_1[Q_{b_1}] | \ldots T | \vec{T}]$$

$$T(\Sigma') = l[n[Q_n] | b_1[Q_{b_1}] | \ldots a[! \text{ out } n.Q | Q | U | \vec{U} | S_1 | \vec{S_1} | \ldots] | T | \vec{T}]$$

- (Open)

$$T(\Sigma) = l[\text{open } n.Q | Q' | n[S | c_1[Q_{c_1}] | \ldots]]$$

$$T(Sigma') = l[Q | Q' | S | c_1[Q_{c_1}] | \ldots]$$

- (BOpen)

$$T(\Sigma) = l[! \text{ open } n.Q | Q' | n[S | c_1[Q_{c_1}] | \ldots]]$$

$$T(Sigma') = l[! \text{ open } n.Q | Q | Q' | S | c_1[Q_{c_1}] | \ldots]$$

- (CapAHeap) La regla sólo mueve un proceso de lugar, de un ambiente al heap del mismo ambiente. $A = (n, \mathcal{P}, H, a)$ y $A' = (n, \mathcal{P}/\{P\}, H\{cap := P : \vec{P}\}, a$, por lo que $T(A) = n[P | P_1 | \ldots | T(a)] \equiv n[P_1 | \ldots | P | T(a)] = T(A')$.

- (BCapAHeap) (NoReaders) (NoWriters) (BNoWriters) Estos casos son idénticos al anterior.

□

3.4.1. Completez

Sería deseable mostrar que la máquina es una implementación completa del cálculo. Esto sin embargo no es posible, dado que el cálculo no posee la propiedad de *confluencia.* Esta propiedad pide que cuando una expresión tenga dos reducciones posibles el resultado final de la evaluación no dependa de cuál sea elegida. En un cálculo que tenga esta propiedad dicha elección se convierte entonces principalmente en una cuestión de eficiencia en la evaluación.

Desafortunadamente, en general los cálculos de procesos carecen de la propiedad de confluencia. Como se ve en el ejemplo de la página 13 la elección de qué subexpresión reducir, afecta de forma irreversible el resultado final. Esto quiere decir que una expresión del cálculo de ambientes tiene más de un resultado posible.

Por otro lado, la máquina virtual es completamente determinista. Esto quiere decir que, dado un estado, siempre hay cuando mucho una transición posible a partir de él. Esto quiere decir que la máquina tiene sólo un resultado posible para cada expresión. De aquí se deduce que la máquina no es una implementación completa del cálculo, puesto que no todas las transiciones legales en este último pueden llegar a ser ejecutadas por la primera.

3.4.2. Ausencia de bloqueos

En vista de la imposibilidad de mostrar completez nos limitaremos a mostrar que la máquina no se detiene, a menos que el proceso del cálculo de ambientes que representa no tenga transiciones posibles. Así podemos asegurar que la máquina progresará en la evaluación siempre que el proceso en el cálculo subyacente no se encuentre detenido.

Esta definición de *detenido* se refiere únicamente a la imposibilidad de progreso, y no a alguna noción de "buena terminación", puesto que carecemos de un criterio para definir esta última.

Teorema 2. *Sea* Σ *un estado de la máquina detenido. Entonces* $\mathsf{T}(\Sigma)$ *es un proceso sin transiciones posibles.*

Demostración. De las reglas de la máquina podemos ver que siempre que un estado tiene por lo menos un proceso en la cola de ejecución hay una regla que se puede aplicar a dicho estado para avanzar un paso, así que un estado de la máquina está detenido si y sólo si su cola de ejecución está vacía.

Procedemos por contradicción. Supongamos que $\Sigma = (R, \mathcal{A})$ está detenido, pero que existe un proceso Q tal que $T(\Sigma) \equiv \xrightarrow{l} Q$. Procedemos por casos sobre la transición l, y exponemos sólo algunos casos representativos.

- *(comm)* Como Q se obtiene a partir de una comunicación tenemos que $T(\Sigma) \equiv C(\langle M \rangle | (x).P | P')$ para algún contexto $C(\dot{})$ y procesos P y P'. Como la cola de ejecución está vacía, tanto $\langle M \rangle$ como $(x).P$ están en el *heap* del ambiente correspondiente. Pero esto no es posible, puesto que un *heap* no puede contener escritores y lectores al mismo tiempo, lo que nos da la contradicción buscada.

- *(in)* $T(\Sigma) = C(m[\text{in}\, n.P | P'] | n[P''])$. Nos fijamos en los procesos $\text{in}\, n.P$ y $n[P'']$. Ambos han sido ejecutados, puesto que la cola de ejecución de Σ está vacía por hipótesis. En particular eso significa que $\text{in}\, n.P$ está en el *heap* de m.

  El hecho de que $\text{in}\, n.P$ esté en un *heap* nos dice que, cuando la máquina lo ejecutó, m no tenía un ambiente hermano de nombre n. Esto significa una de dos cosas: El proceso $n[P'']$ todavía no se había ejecutado, o el ambiente n no era hermano de m en ese momento.

  En el primer caso, al ejecutarse $n[P'']$ la regla *NewAmb* hubiera ejecutado el $\text{in}\, n.P$, que estaba en el *heap* de m.

  En el segundo caso, alguna regla movió a n desde su posición anterior a la de hermano de m. Esta misma regla hubiera trasladado a $\text{in}\, n.P$ desde el *heap* hasta la cola de ejecución.

  Por lo tanto, ninguno de los dos casos es posible, lo que nos da la contradicción buscada.

  Los casos de las otras dos capacidades, *out* y *open*, así como de las reglas para ! son completamente análogos. □

Capítulo 4

Implementación

En este capítulo damos una visión general de la implementación en Haskell [24] de la máquina virtual. Gracias a las características del lenguaje usado esta implementación es poco más que una transliteración de las reglas dadas. El código fuente completo se incluye en el apéndice B.

La implementación está organizada alrededor del módulo *Maquina*. En éste se define la representación de los estados de la máquina, que consisten en una cola de (identificadores de) procesos, un árbol de ambientes y una lista que asocia los identificadores con cada proceso. Hay además un contador que se incrementa cada que se introduce un nuevo nombre con el operador ν.

```
type Rqueue = [Pid]
data Mach = Mach { r :: Rqueue,
                   t :: Arbol.Arbol (Ambient Pid),
                   p :: Plist,
                   next_name :: Int}
```

El estado inicial de la máquina tiene un único proceso en la cola, un árbol con un único ambiente y el contador de nombres en 1. El proceso inicial es un dato de tipo `Proc`, definido en el módulo *Proc*.

El módulo define una lista de reglas. Cada una de estas reglas consta de un nombre, una condición y una acción. El nombre se usa sólo para informar al usuario sobre la regla elegida. La condición es una función booleana sobre los estados, que determina si la regla es aplicable al estado dado. La acción es una función de estados a estados, que determina la transformación dada

por esta regla.

```
data Regla a = R { nombre :: String, cond :: a -> Bool, act :: a -> a}
```

La Máquina avanza buscando la regla siguiente dado su estado actual y aplicándola a dicho estado. La lista de nombres está indexada por los tipos de procesos. La *regla siguiente* se define como la primera aplicable al tipo de proceso que está al inicio de la cola y tal que su condición es verdadera.

```
avanzaMaquina m = (act . siguienteRegla) m m

siguienteRegla m = head [ r | r <- siguientesReglas, cond r m]
    where
    siguientesReglas = obtenRegla reglas (fst (getproc (p m) firstpid))
    firstpid = head (r m)
```

El resto del módulo define las reglas.

Para su operación el módulo *Maquina* utiliza los servicios de varios otros. Son de particular interés los módulos *Ambient*, *Arbol*, *Proc* y *Heap*. Cada uno de éstos define estructuras de datos y sus operaciones.

Arbol define un tipo de datos genérico de árboles n-arios. Un árbol es una colección de identificadores de nodos, que a su vez son cuartetas de un identificador para el padre, el propio, la lista de hijos y un dato arbitrario. El módulo define funciones genéricas para modificar y recorrer un árbol.

```
data Eq a => Node a = Node { parent   :: Aid,
                             self     :: Aid,
                             datum    :: a,
                             children :: [Aid] }

data Arbol a = Arbol { alist :: (AList.AList Aid (Node a)),
                       lastkey :: Aid }
```

Los ambientes constan de un nombre, una lista de objetos que en nuestro caso son identificadores de procesos, el identificador de su posición en el árbol y un Heap. El módulo define funciones para obtener hijos, hermanos y padres con un nombre dado, así como funciones para añadir hijos y para disolver un ambiente.

```
data Show a => Ambient a = Amb { name :: String,
                                 procs :: [a],
                                 aid :: Arbol.Aid,
```

```
heap :: Heap.Heap String a }
```

Los Heaps son implementados en el módulo del mismo nombre y consisten simplemente de 4 listas de pares llave-valor. Las llaves son, en nuestro caso, cadenas que representan el nombre del ambiente que tiene bloqueados a algunos procesos. El valor es la lista de dichos procesos.

El resto de los módulos del sistema son simplemente funciones de apoyo, como la implementación de las listas llave-valor o la interfaz del sistema con la línea de comandos.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

Las contribuciones principales de este trabajo son las siguientes:

- Hemos concluido que el modelo teórico más adecuado para el cómputo móvil de entre los existentes en la literatura es el cálculo de ambientes, debido a que fue diseñado tomando en cuenta las restricciones características de la movilidad. Al mismo tiempo es un modelo sencillo que introduce pocos conceptos nuevos.

- Se diseño una máquina virtual para ejecutar el cálculo de ambientes. Se demostró que la máquina implementa correctamente la semántica del cálculo y que es tan completa como la implementación determinista lo permite. El diseño de la máquina descubrió una característica poco conveniente del cálculo que se remedió por medio de la introducción de la replicación acotada.

- Se implementó esta máquina virtual. Esta implementación de referencia permite ver que el diseño propuesto es viable. Abre además el camino para la experimentación, en la implementación en sí, en el diseño de la máquina y en el uso práctico del cálculo de ambientes.

- Una versión abreviada de este trabajo fue aceptada en la *2nd International Conference on Electrical and Electronics Engineering and*

XI Conference on Electrical Engineering, conferencia arbitrada del Departamento de Ingeniería Eléctrica de CINVESTAV y la IEEE.

5.2. Trabajo futuro

Presentamos algunas rutas de trabajo futuro posibles.

- Buscar los puntos en los que sea posible optimizar el desempeño de la implementación. Todas las estructuras de datos están implementadas tomando como principal criterio la sencillez. Implementaciones más eficientes de las mismas podrían impactar significativamente el desempeño de la máquina.
- El diseño de la máquina no especifica una estrategia de evaluación para los casos en que hay más de una reducción posible para un estado. En la implementación presentada dicha elección se hace de forma implícita a través del orden en que los procesos son colocados en la cola de ejecución. Si se modifica ese orden es posible experimentar con diversas estrategias, lo que permitiría indagar acerca de sus propiedades.
- Modificar el diseño de la máquina para adecuarla a una implementación distribuida. Si bien el diseño actual procura realizar sólo acciones locales, no se tomaron en cuenta las necesidades de bloqueo y sincronización que una implementación distribuida tendría.
- Diseñar e implementar una sintaxis para el cálculo que haga más fácil experimentar con el uso del lenguaje en situaciones prácticas.
- Utilizar la máquina virtual como plataforma de experimentación para diseñar un lenguaje de programación para el cómputo móvil.
- Modificar el diseño para explorar versiones modificadas del CA, como CAC, Safe Ambients, sistemas de tipos, etc.

Apéndice A

Ejemplos

En este apéndice presentamos ejemplos del cálculo de ambientes, ejecutados por la máquina virtual.

A.1. RPC

Este ejemplo implementa una llamada a un procedimiento remoto. La codificación original en el CA está tomada del artículo de Cardelli y Gordon [12].

El siguiente proceso envía un argumento a al proceso que se ejecuta en un cierto ambiente y recupera la respuesta de ese proceso sobre el nombre x. M denota una ruta desde el ambiente actual hasta el ambiente destino y M^{-1} denota la ruta de vuelta.

$$(\nu n)(io[M.(< a > | \text{open } res.(x).n[M^{-1}. < x >])] | \text{open } n) | (x).P$$

El ambiente destino debe tener la forma

$$m[! \text{open } io | !(a).Q.res[< x >]]$$

En donde Q es el proceso que calcula el resultado deseado. Obsérvese que si hubiera necesidad de tener varios procedimientos remotos, cada uno tendría que estar en un ambiente separado. Además, Q no puede estar formado por varios procesos paralelos que se comuniquen, a menos que se les coloque

dentro de un ambiente propio, puesto que no se puede mantener separada dicha comunicación interna a Q de las peticiones de clientes.

Para fijar ideas hacemos $(x).P = 0$, $Q = 0$, colocamos el proceso que desea comunicarse dentro de un ambiente k y hacemos a m un hermano de k. Eliminamos también el ! del proceso $!(a).Q.res[< x >]$, para simplificar. Así, $M = \text{out}\, k.\, \text{in}\, m$, $M^{-1} = \text{out}\, m.\, \text{in}\, k$ y la ejecución del proceso se ve como en la figura A.1.

Máquina virtual La siguiente traza se obtuvo al ejecutar el comando

```
./boxed k[ (nu n)(io[out k.in m.(<arg>|open res.(x).n[out m.in k.<x>])] |
        open n.(0)) ] | m[ (!open io.(0)) | (in).(res[<in>]) ]
```

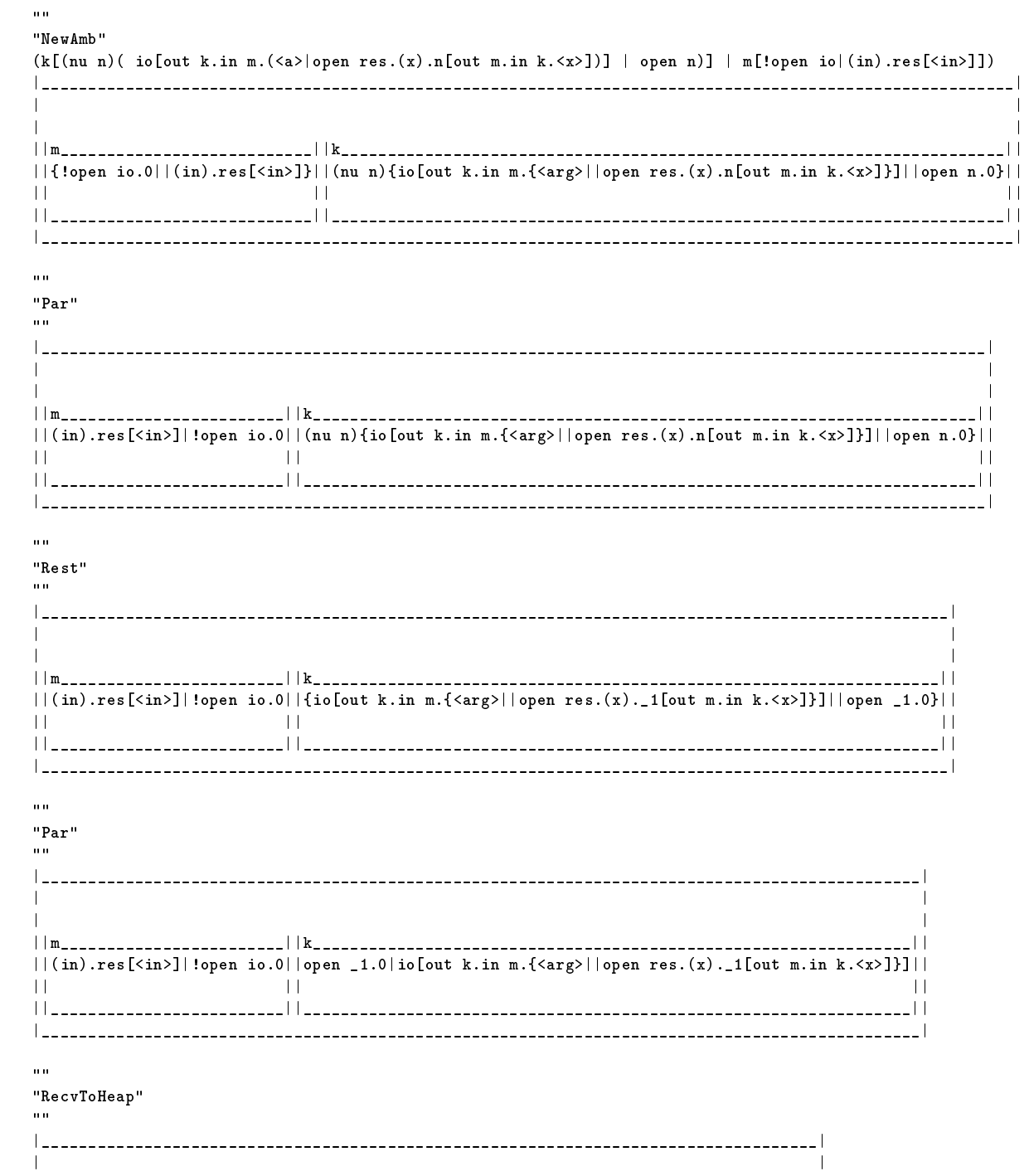
Obsérvese cómo la máquina implementa la creación del nombre privado n sustituyéndolo por el nombre _1 que es un nombre que, de acuerdo a la gramática, no puede aparecer en el proceso a ejecutar.

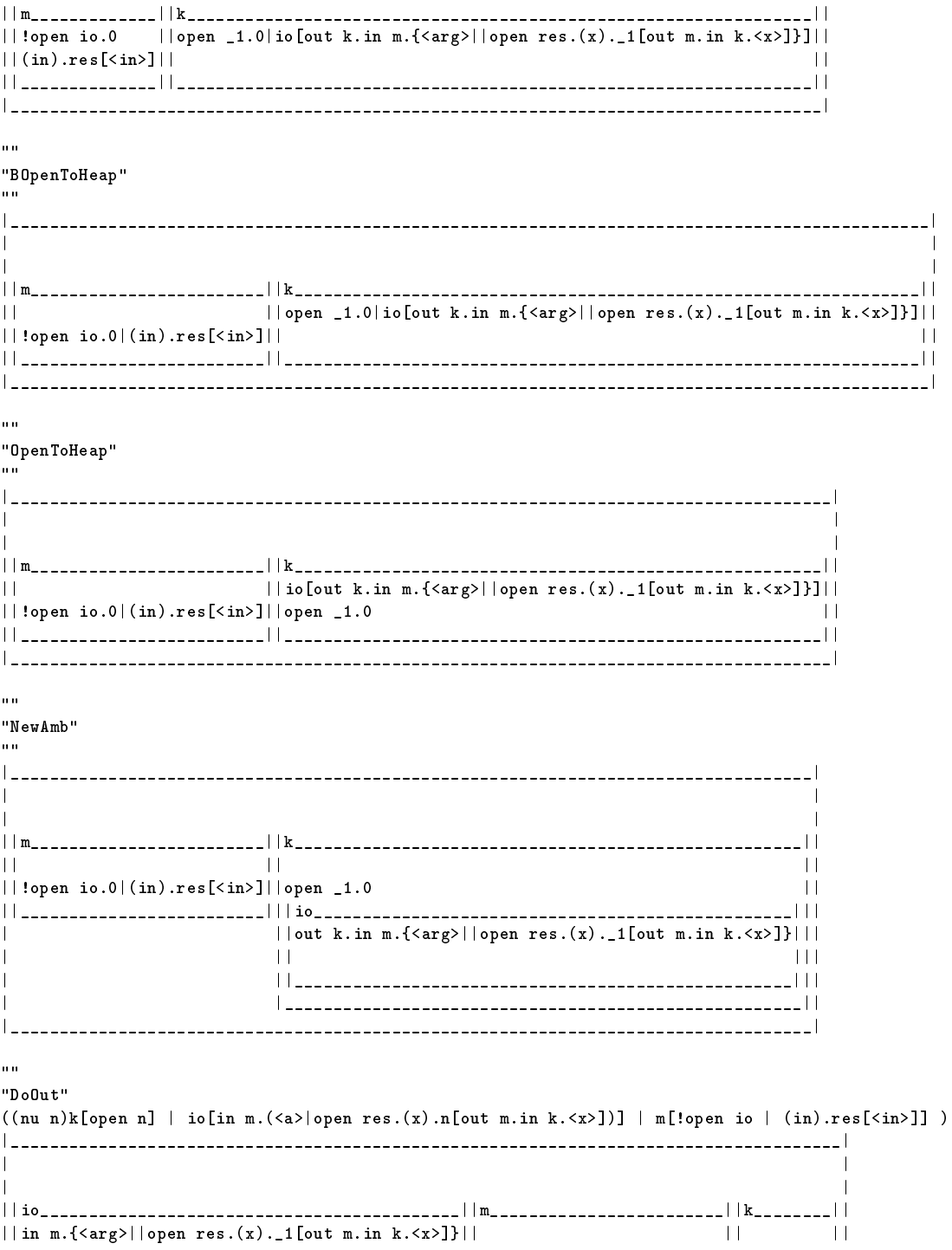
Algunas cajas tienen, arriba y entre paréntesis, una expresión de proceso del cálculo de ambientes. Dichas expresiones corresponden a la ejecución manual presentada en la figura A.1 y fueron añadidas manualmente.

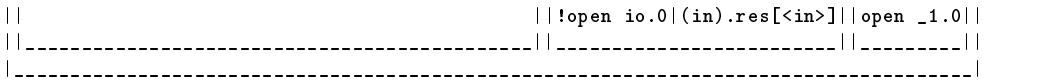
```
|-------------------------------------------------------------------------------------------------------------|
|{k[(nu n){io[out k.in m.{<arg>||open res.(x).n[out m.in k.<x>]}]||open n.0}]||m[{!open io.0||(in).res[<in>]}]}|
|                                                                                                             |
|-------------------------------------------------------------------------------------------------------------|

""
"Par"
""
|-------------------------------------------------------------------------------------------------------------|
|m[{!open io.0||(in).res[<in>]}]|k[(nu n){io[out k.in m.{<arg>||open res.(x).n[out m.in k.<x>]}]||open n.0}]|
|                                                                                                             |
|-------------------------------------------------------------------------------------------------------------|

""
"NewAmb"
""
|-----------------------------------------------------------------------|
|k[(nu n){io[out k.in m.{<arg>||open res.(x).n[out m.in k.<x>]}]||open n.0}]|
|                                                                       |
||m_________________________|                                          |
||{!open io.0||(in).res[<in>]}|                                        |
||                           |                                          |
||___________________________|                                          |
|-----------------------------------------------------------------------|
```

$$
\begin{array}{ll}
& \mathrm{k}[(\nu \mathrm{n})(\mathrm{io}[\mathrm{out\ k.\ in\ m.}(\langle \mathrm{a} \rangle | \\
& \quad \mathrm{open\ res.}(\mathrm{x}).\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{x} \rangle])] | \ \mathrm{open\ n})] | & \text{(A.1)} \\
& \mathrm{m}[!\ \mathrm{open\ io} | \ (\mathrm{in}).\mathrm{res}[\langle \mathrm{in} \rangle]] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{io}[\mathrm{in\ m.}(\langle \mathrm{a} \rangle | \ \mathrm{open\ res.}(\mathrm{x}).\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{x} \rangle])] | & \text{(A.2)} \\
& \quad \mathrm{m}[!\ \mathrm{open\ io} | \ (\mathrm{in}).\mathrm{res}[\langle \mathrm{in} \rangle]] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{m}[\mathrm{io}[\langle \mathrm{a} \rangle | \ \mathrm{open\ res.}(\mathrm{x}).\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{x} \rangle]] | & \text{(A.3)} \\
& \quad\quad !\ \mathrm{open\ io} | \ (\mathrm{in}).\mathrm{res}[\langle \mathrm{in} \rangle]] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{m}[\langle \mathrm{a} \rangle | \ \mathrm{open\ res.}(\mathrm{x}).\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{x} \rangle] | & \text{(A.4)} \\
& \quad\quad !\ \mathrm{open\ io} | \ (\mathrm{in}).\mathrm{res}[\langle \mathrm{in} \rangle]] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{m}[\mathrm{res}[\langle \mathrm{a} \rangle] | \ \mathrm{open\ res.}(\mathrm{x}).\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{x} \rangle] | & \text{(A.5)} \\
& \quad\quad !\ \mathrm{open\ io}] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{m}[\langle \mathrm{a} \rangle | \ (\mathrm{x}).\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{x} \rangle] | & \text{(A.6)} \\
& \quad\quad !\ \mathrm{open\ io}] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{m}[\mathrm{n}[\mathrm{out\ m.\ in\ k.}\langle \mathrm{a} \rangle] | & \text{(A.7)} \\
& \quad\quad !\ \mathrm{open\ io}] \\
& \quad (\nu \mathrm{n})\mathrm{k}[\mathrm{open\ n}] | \\
\longrightarrow & \quad \mathrm{m}[!\ \mathrm{open\ io}] | & \text{(A.8)} \\
& \quad\ \mathrm{n}[\mathrm{in\ k.}\langle \mathrm{a} \rangle] \\
\longrightarrow & (\nu \mathrm{n})\mathrm{k}[\mathrm{n}[\langle \mathrm{a} \rangle] | \ \mathrm{open\ n}] | \\
& \quad \mathrm{m}[!\ \mathrm{open\ io}] & \text{(A.9)} \\
\longrightarrow & (\nu \mathrm{n})\mathrm{k}[\langle \mathrm{a} \rangle] | \ \mathrm{m}[!\ \mathrm{open\ io}] & \text{(A.10)}
\end{array}
$$

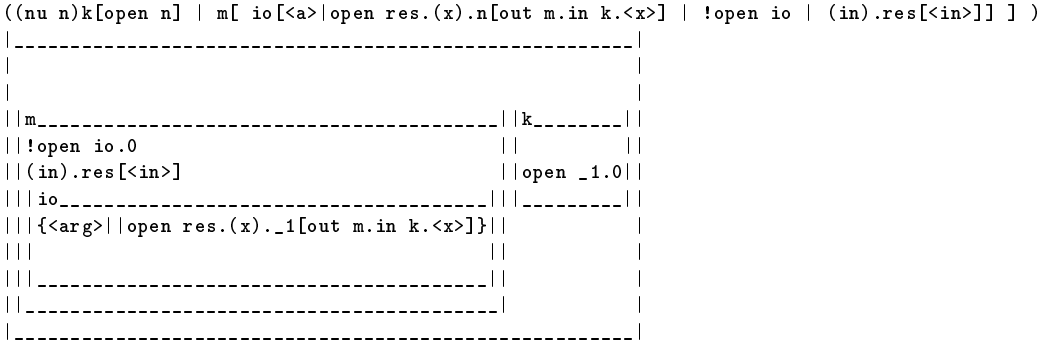
Figura A.1: Ejecución manual del ejemplo RPC


""
"NewAmb"
(k[(nu n)(io[out k.in m.(<a>|open res.(x).n[out m.in k.<x>])] | open n)] | m[!open io|(in).res[<in>]])
||m||k||
||{!open io.0||(in).res[<in>]}||(nu n){io[out k.in m.{<arg>||open res.(x).n[out m.in k.<x>]}]||open n.0}||
""
"Par"
""
||m||k||
||(in).res[<in>]||!open io.0||(nu n){io[out k.in m.{<arg>||open res.(x).n[out m.in k.<x>]}]||open n.0}||
""
"Rest"
""
||m||k||
||(in).res[<in>]||!open io.0||{io[out k.in m.{<arg>||open res.(x)._1[out m.in k.<x>]}]||open _1.0}||
""
"Par"
""
||m||k||
||(in).res[<in>]||!open io.0||open _1.0|io[out k.in m.{<arg>||open res.(x)._1[out m.in k.<x>]}]||
""
"RecvToHeap"
""

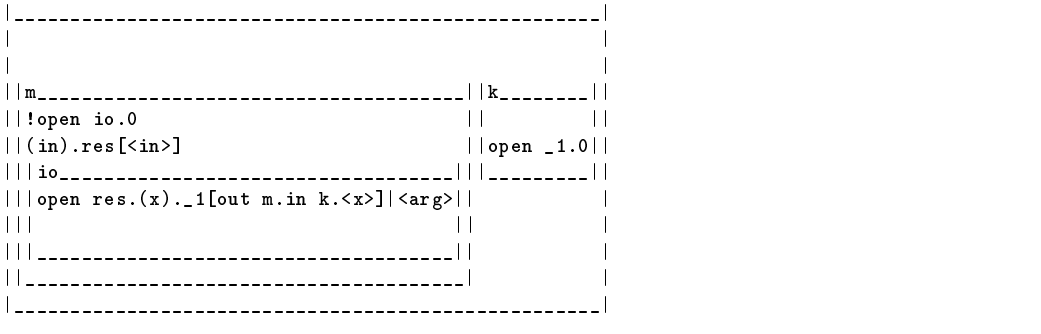

	m_____________		k____	
	!open io.0		open _1.0	io[out k.in m.{<arg>		open res.(x)._1[out m.in k.<x>]}]	
	(in).res[<in>]			
	_______________		____	
""
"BOpenToHeap"
""
	m_____________________		k____	
	||open _1.0|io[out k.in m.{<arg>||open res.(x)._1[out m.in k.<x>]}]|
	!open io.0	(in).res[<in>]			
	_______________________		____	
""
"OpenToHeap"
""
	m_____________________		k____	
			io[out k.in m.{<arg>		open res.(x)._1[out m.in k.<x>]}]	
	!open io.0	(in).res[<in>]		open _1.0	
	_______________________		____	
""
"NewAmb"
""
	m_____________________		k____	
	!open io.0	(in).res[<in>]		open _1.0	
	_______________________			io_____________________________________		
	out k.in m.{<arg>		open res.(x)._1[out m.in k.<x>]}		
	_____________________________________		
_____________________________________	
""
"DoOut"
((nu n)k[open n] | io[in m.(<a>|open res.(x).n[out m.in k.<x>])] | m[!open io | (in).res[<in>]])
||io_____________________________________||m_____________________||k_______||
||in m.{<arg>||open res.(x)._1[out m.in k.<x>]}|| || ||



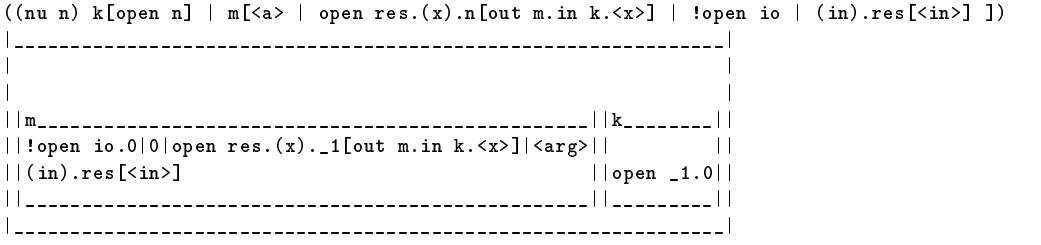
""
"DoIn"



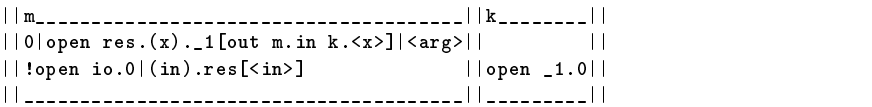
""
"Par"
""



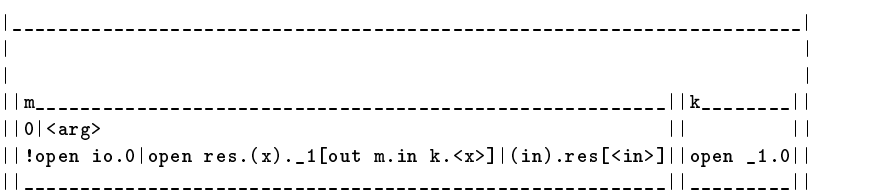
""
"DoBOpen"



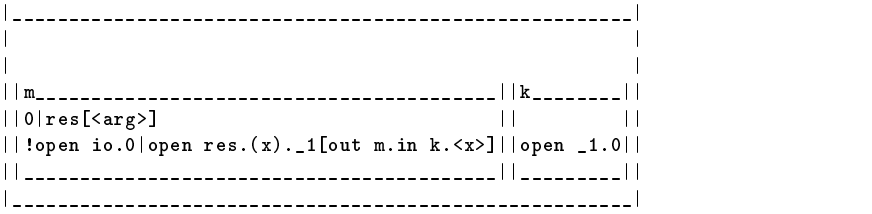
""
"BOpenToHeap"
""



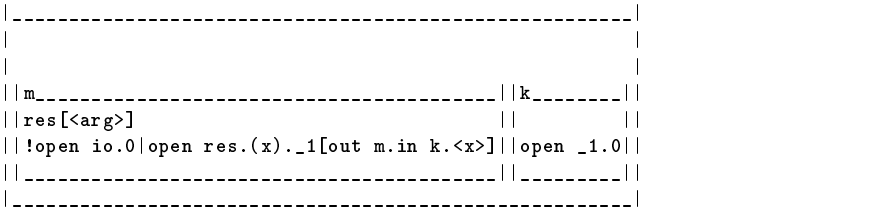
""
"OpenToHeap"
""



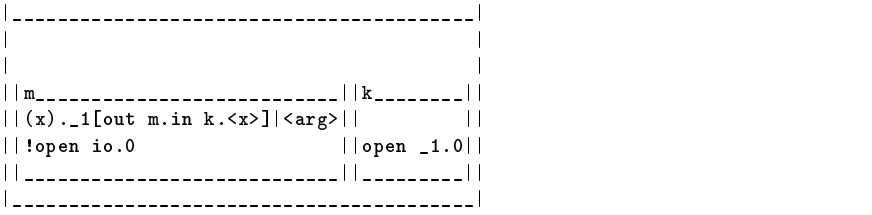
""
"Send"
((nu n) k[open n] | m[res[<a>] | open res.(x).n[out m.in k.<x>] | !open io])



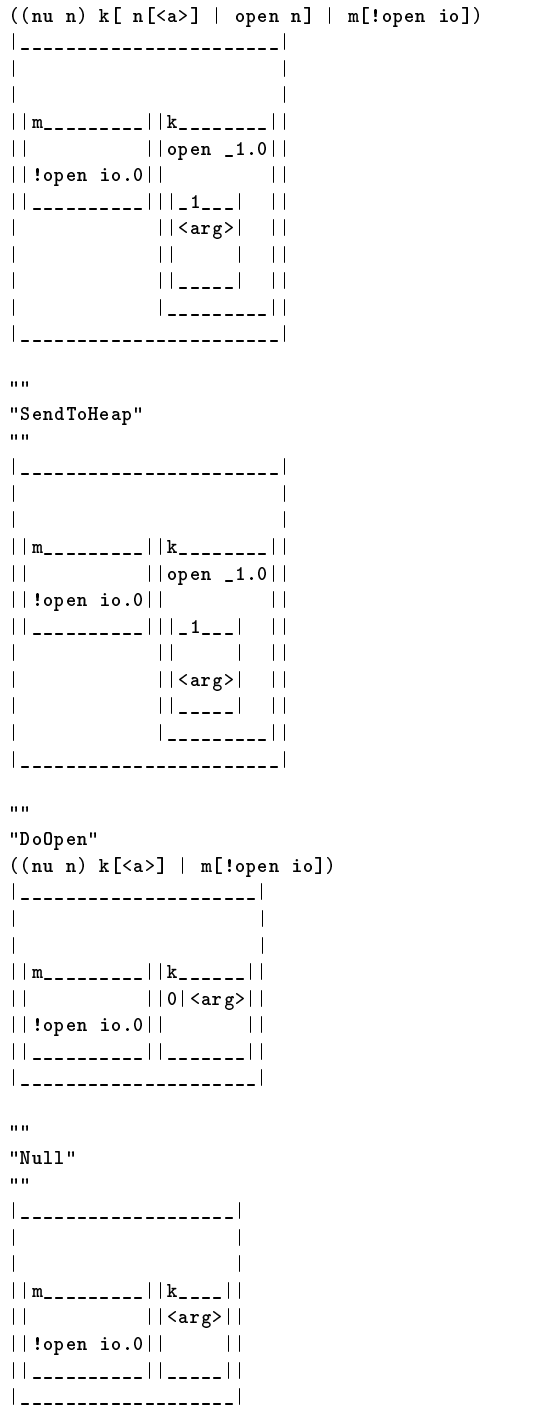
""
"Null"
""



""
"OpenNewAmb"
((nu n) k[open n] | m[<a> | (x).n[out m.in k.<x>] | !open io])




""
"SendToHeap"
""
|_____________________________________|
| |
| |
	m____________________		k________	
	(x)._1[out m.in k.<x>]			
	!open io.0	<arg>		open _1.0	
	_____________________		_________	
____
""
"Recv"
((nu n) k[open n] | m[n[out m.in k.<a>] | !open io])
|_____________________________________|
| |
| |
	m___________________		k________	
	_1[out m.in k.<arg>]			
	!open io.0		open _1.0	
	____________________		_________	
____
""
"NewAmb"
""
|_______________________________|
| |
| |
	m_________________		k________	
	!open io.0		open _1.0	
		_1_____________			_________	
		out m.in k.<arg>		
		_________________		
	__________________	
____
""
"DoOut"
((nu n) k[open n] | n[in k.<a>] | m[!open io])
|_____________________________________|
| |
| |
	_1________		m_________		k________	
	in k.<arg>					
			!open io.0		open _1.0	
	___________		___________		_________	
____
""
"DoIn"


((nu n) k[n[<a>] | open n] | m[!open io])
m
!open io.0
k
open _1.0
_1
<arg>
""
"SendToHeap"
""
m
!open io.0
k
open _1.0
_1
<arg>
""
"DoOpen"
((nu n) k[<a>] | m[!open io])
m
!open io.0
k
0|<arg>
""
"Null"
""
m
!open io.0
k
<arg>

```
""
"SendToHeap"
""
|-------------------|
|                   |
|                   |
||m_________||k____||
||          ||     ||
||!open io.0||<arg>||
||__________||_____||
|-------------------|

""
""
""
```

A.2. Firewall

Otro ejemplo tomado de Cardelli[12]. Éste muestra la posibilidad de llevar a cabo controles de acceso.

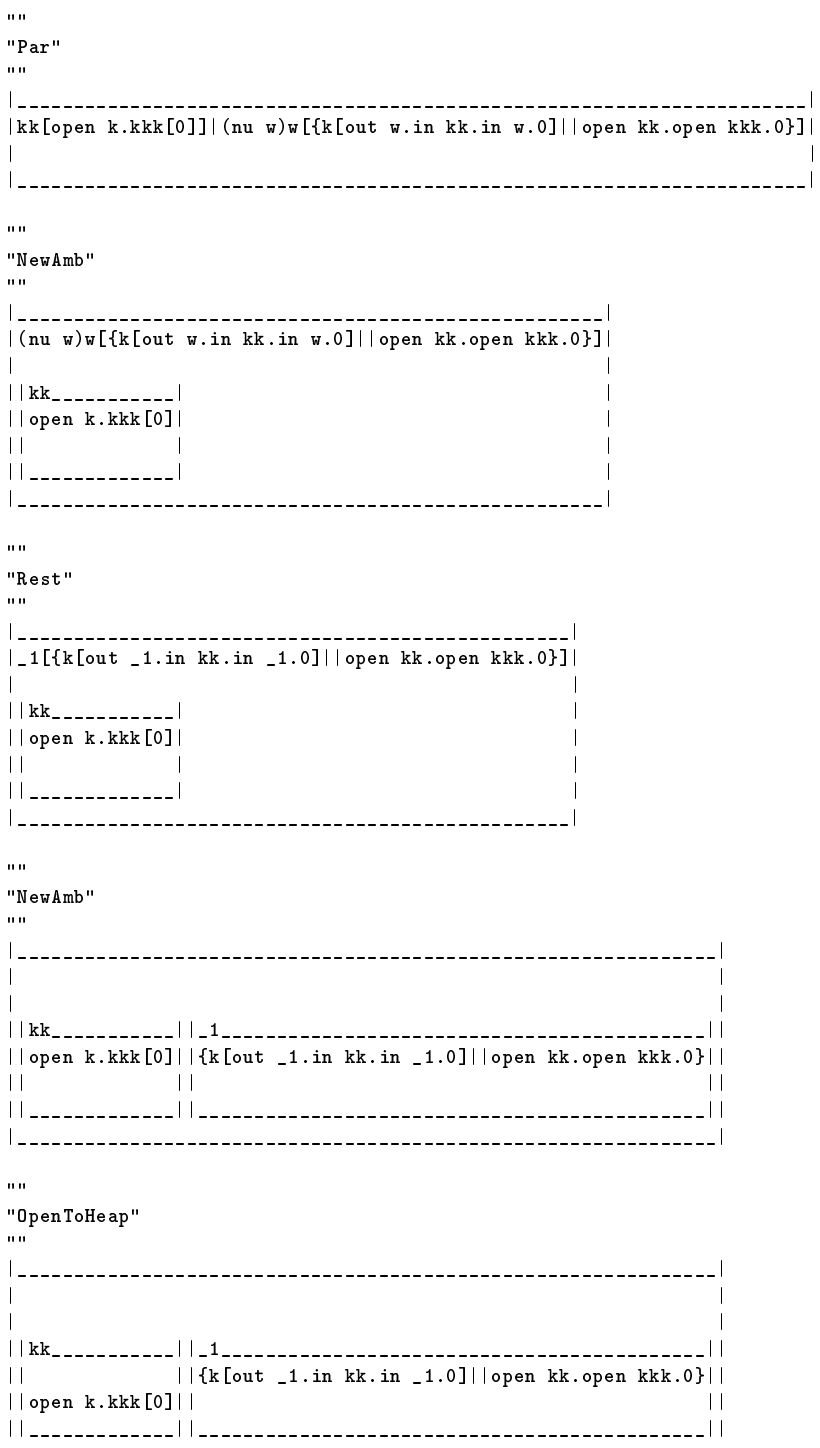
Las "contraseñas" en este caso son tres nombres k, k' y k''. El ambiente w mantiene su nombre en secreto. Para permitir la entrada a un ambiente cliente manda un "piloto" $k[\text{out}\, w.\, \text{in}\, k'.\text{in}w]$. La ejecución de $\text{in}\, k'$ demuestra que el agente conoce la contraseña k'. El agente se asegura de estar interactuando con w por que el piloto se llama k. El tercer ambiente k'' se usa para confinar el contenido del agente y evitar que interfiera con el protocolo.

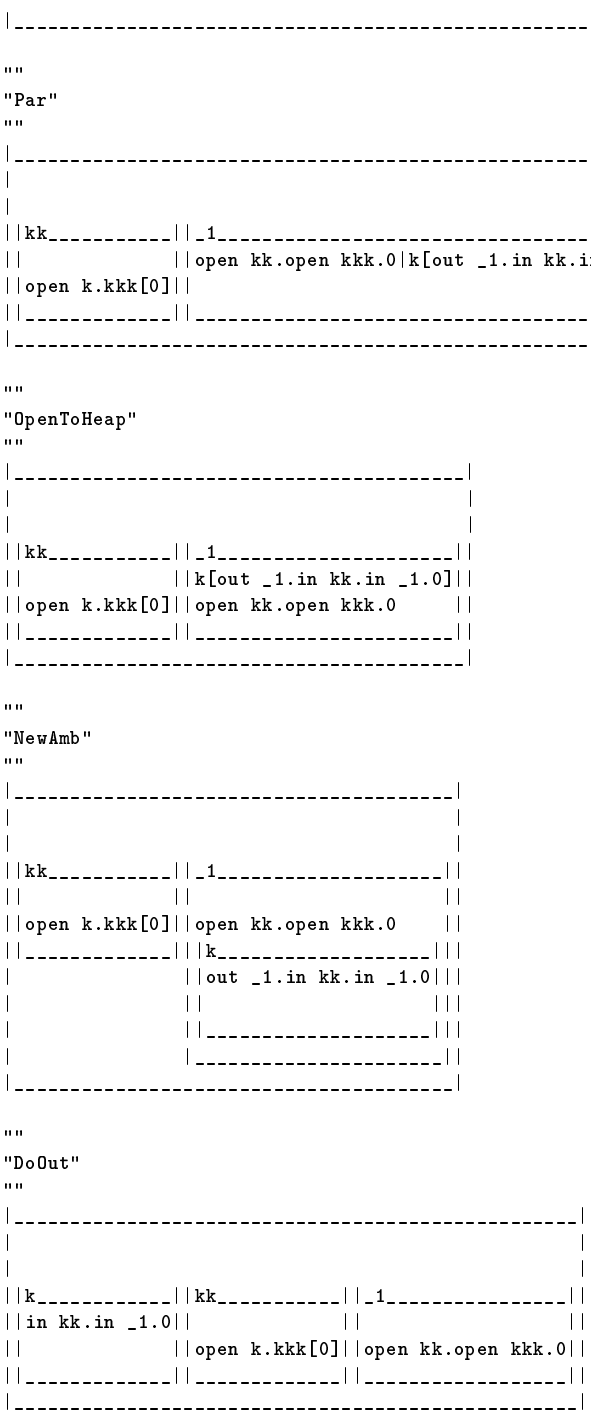
$$Firewall := (\nu w)w[k[\text{out}\, w.\, \text{in}\, k'.\text{in}w] \mid \text{open}k'.\text{open}k''.P]$$
$$Agente := k'[\text{open}k.k''[Q]]$$

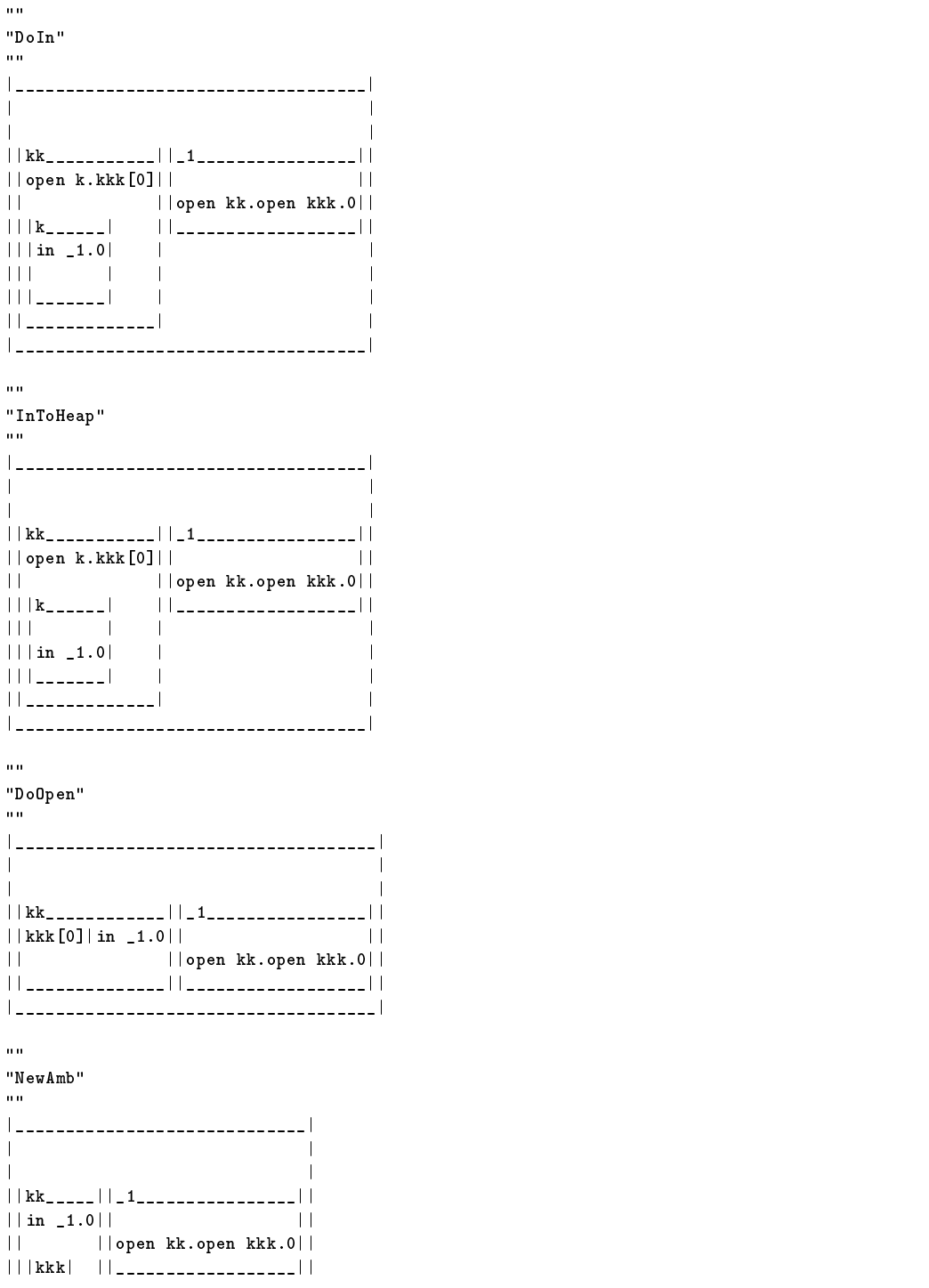
El comando ejecutado fue el siguiente:

```
./boxed (nu w)(w[ k[out w.in kk.in w.(0)] | open kk.open kkk.(0) ])|
         kk[open k.(kkk[0])]
```

```
|-------------------------------------------------------------------------|
|{(nu w)w[{k[out w.in kk.in w.0]||open kk.open kkk.0}]||kk[open k.kkk[0]]}|
|                                                                         |
|-------------------------------------------------------------------------|
```


""
"Par"
""
|kk[open k.kkk[0]]|(nu w)w[{k[out w.in kk.in w.0]|open kk.open kkk.0}]|
""
"NewAmb"
""
|(nu w)w[{k[out w.in kk.in w.0]|open kk.open kkk.0}]|
|kk_____|
|open k.kkk[0]|
""
"Rest"
""
|_1[{k[out _1.in kk.in _1.0]|open kk.open kkk.0}]|
|kk_____|
|open k.kkk[0]|
""
"NewAmb"
""
|kk_____||_1_____|
|open k.kkk[0]||{k[out _1.in kk.in _1.0]|open kk.open kkk.0}|
""
"OpenToHeap"
""
|kk_____||_1_____|
|{k[out _1.in kk.in _1.0]|open kk.open kkk.0}|
|open k.kkk[0]|


""
"Par"
""
kk
_1
open kk.open kkk.0|k[out _1.in kk.in _1.0]
open k.kkk[0]
""
"OpenToHeap"
""
kk
_1
k[out _1.in kk.in _1.0]
open k.kkk[0]
open kk.open kkk.0
""
"NewAmb"
""
kk
_1
open k.kkk[0]
open kk.open kkk.0
k
out _1.in kk.in _1.0
""
"DoOut"
""
k
kk
_1
in kk.in _1.0
open k.kkk[0]
open kk.open kkk.0


""
"DoIn"
""
kk
open k.kkk[0]
k
in _1.0
_1
open kk.open kkk.0
""
"InToHeap"
""
kk
open k.kkk[0]
k
in _1.0
_1
open kk.open kkk.0
""
"DoOpen"
""
kk
kkk[0]
in _1.0
_1
open kk.open kkk.0
""
"NewAmb"
""
kk
in _1.0
_1
open kk.open kkk.0
kkk

```
|||0  |  |                   |
|||   |  |                   |
|||___|  |                   |
||_______|                   |
|_____________________________|

""
"DoIn"
""
|____________________|
|                    |
|                    |
||_1_________________||
||open kk.open kkk.0||
||                  ||
|||kk___|           ||
|||0    |           ||
|||     |           ||
||||kkk||           ||
||||0  ||           ||
||||   ||           ||
||||___||           ||
|||_____|           ||
||__________________||
|____________________|

""
"Null"
""
|____________________|
|                    |
|                    |
||_1_________________||
||open kk.open kkk.0||
||                  ||
|||kk___|           ||
|||     |           ||
|||     |           ||
||||kkk||           ||
||||0  ||           ||
||||   ||           ||
||||___||           ||
|||_____|           ||
||__________________||
|____________________|

""
"Null"
""
|____________________|
|                    |
|                    |
||_1_________________||
||open kk.open kkk.0||
```

```
||                  ||
|||kk___|           ||
|||     |           ||
|||     |           ||
||||kkk||           ||
||||   ||           ||
||||   ||           ||
||||___||           ||
|||_____|           ||
||__________________||
|____________________|

""
"DoOpen"
""
|____________|
|            |
|            |
||_1________||
||open kkk.0||
||          ||
|||kkk|     ||
|||   |     ||
|||   |     ||
|||___|     ||
||__________||
|____________|

""
"DoOpen"
""
|____|
|    |
|    |
||_1||
||0 ||
||  ||
||__||
|____|

""
"Null"
""
|____|
|    |
|    |
||_1||
||  ||
||  ||
||__||
|____|

""
""
```

""

Apéndice B

Código fuente

Este apéndice presenta el código fuente de la implementación en su totalidad. Este código ha sido probado con el intérprete Hugs y con el compilador GHC 6.2, ambos sobre plataforma Linux.

B.1. Analizador sintáctico

Este módulo implementa el análisis sintáctico del lenguaje fuente de la máquina.

```
module Parse (parseString) where
import qualified List
import Char
import Proc
```

Los lexemas válidos son [,], |, (,), <, >, '.', ! e identificadores, que se representan internamente por constantes del tipo enumerado `Token`.

```
data Token = RBrak | LBrak | Bar | RPar | LPar
           | RAng | LAng | Dot | Bang | Id String
             deriving (Eq, Show)

lexer [] = []
lexer cs = tok : (lexer rest)
    where
    (tok, rest) = oneLex cs
    oneLex ('[':cs) = (LBrak, cs)
    oneLex (']':cs) = (RBrak, cs)
```

```
    oneLex ('|':cs) = (Bar, cs)
    oneLex ('(':cs) = (LPar, cs)
    oneLex (')':cs) = (RPar, cs)
    oneLex ('<':cs) = (LAng, cs)
    oneLex ('>':cs) = (RAng, cs)
    oneLex ('.':cs) = (Dot, cs)
    oneLex ('!':cs) = (Bang, cs)
    oneLex (c:cs) | isSpace c = oneLex cs
    oneLex ('0':cs) = (Id "0", cs)
    oneLex cs = ( Id (takeWhile isAlpha cs), dropWhile isAlpha cs)
```

La función parser toma una lista de lexemas y produce como resultado un proceso.

```
parser toks = proc
    where
    candidates = [ ps | (ps, []) <- pars ]
    proc = if null candidates then
           error "Error de sintaxis"
           else head candidates
    pars = parseProc toks
```

parseString combina los analizadores léxico y sintáctico y produce un proceso a partir de una cadena de caracteres.

```
parseString = parser . lexer
```

Los analizadores básicos para expresiones

```
parseBasicExp = pAlts [parseName, parseIn, parseOut, parseOpen]
parseExp = pAlts [parseName, parsePath]

parseName (Id name : ts) = [(Name name, ts)]
parseName _ = []

parseIn = pThen mkIn (pTok (Id "in") pUn) parseName
    where mkIn _ name = In name
parseOut = pThen mkOut (pTok (Id "out") pUn) parseName
    where mkOut _ name = Out name
parseOpen = pThen mkOpen (pTok (Id "open") pUn) parseName
    where mkOpen _ name = Open name

parsePath toks = [ (Path es, t) | (es, t) <- oneOrMoreSepBy Dot
                   parseBasicExp toks]
```

Los analizadores para procesos.

```
procParsers = parsePar:basicProcParsers
basicProcParsers = [parseNull, parseParen, parseRest,
                    parseSend, parseRecv, parseAmb,
                    parseSeq, parseBangMove,
                    parseBangRecv]

parseProc = pAlts procParsers
parseBasicProc = pAlts basicProcParsers

parsePar toks = [ (Par ps, t) | (ps, t) <- twoOrMoreSepBy Bar
                  parseBasicProc toks]

parseNull = pTok (Id "0") Null

parseParen = pThen3 keepProc (pTok LPar pUn) parseProc (pTok RPar pUn)
    where keepProc _ p _ = p

parseRest = pThen5 mkRest (pTok LPar pUn) (pTok (Id "nu") pUn)
            parseExp (pTok RPar pUn) parseBasicProc
    where mkRest _ _ e _ p = Rest e p

parseSend = pThen3 mkSend (pTok LAng pUn) parseExp (pTok RAng pUn)
    where mkSend _ e _ = Send e

parseRecv = pThen5 mkRecv (pTok LPar pUn) parseName
            (pTok RPar pUn) (pTok Dot pUn) parseBasicProc
    where mkRecv _ e _ _ p = Recv e p

parseAmb = pThen4 mkAmb parseName (pTok LBrak pUn) parseProc (pTok RBrak pUn)
    where mkAmb n _ p _ = Amb n p

parseSeq = pThen3 mkSeq parseExp (pTok Dot pUn) parseBasicProc
    where mkSeq e _ p = Seq e p

parseBangMove = pThen4 mkBMove (pTok Bang pUn) parseExp (pTok Dot pUn)
                parseBasicProc
    where mkBMove _ e _ p = BangMove e p

parseBangRecv = pThen6 mkBRecv (pTok Bang pUn) (pTok LPar pUn) parseName
            (pTok RPar pUn) (pTok Dot pUn) parseBasicProc
```

```
    where mkBRecv _ _ e _ _ p = BangRecv e p
```

Las siguientes son funciones auxilares.

```
pAlt p1 p2 toks = (p1 toks) ++ (p2 toks)

pAlts ps toks = concat [ p toks | p <- ps ]

pThen comb p1 p2 toks = [ (comb r1 r2, t2) |
                          (r1, t1) <- p1 toks,
                          (r2, t2) <- p2 t1 ]

pThen3 comb p1 p2 p3 toks = [ (comb r1 r2 r3, t3) |
                              (r1, t1) <- p1 toks,
                              (r2, t2) <- p2 t1,
                              (r3, t3) <- p3 t2]

pThen4 comb p1 p2 p3 p4 toks = [ (comb r1 r2 r3 r4, t4) |
                                 (r1, t1) <- p1 toks,
                                 (r2, t2) <- p2 t1,
                                 (r3, t3) <- p3 t2,
                                 (r4, t4) <- p4 t3]

pThen5 comb p1 p2 p3 p4 p5 toks = [ (comb r1 r2 r3 r4 r5, t5) |
                                    (r1, t1) <- p1 toks,
                                    (r2, t2) <- p2 t1,
                                    (r3, t3) <- p3 t2,
                                    (r4, t4) <- p4 t3,
                                    (r5, t5) <- p5 t4]

pThen6 comb p1 p2 p3 p4 p5 p6 toks = [ (comb r1 r2 r3 r4 r5 r6, t6) |
                                       (r1, t1) <- p1 toks,
                                       (r2, t2) <- p2 t1,
                                       (r3, t3) <- p3 t2,
                                       (r4, t4) <- p4 t3,
                                       (r5, t5) <- p5 t4,
                                       (r6, t6) <- p6 t5]

pUn = undefined
pEmpty r toks = [(r, toks)]

pTok _ _ [] = []
pTok tok r (t:ts) | t == tok = [(r,ts)]
```

```
                  | True      = []

zeroOrMore p = (oneOrMore p) `pAlt` (pEmpty [])
oneOrMore p = p `pThen'` (zeroOrMore p)
    where pThen' = pThen (:)

oneOrMoreSepBy tok p toks = keepLongest ((p `pThen'` (zeroOrMore pair)) toks)
    where
    pThen' = pThen (:)
    pair = pThen keepSnd (pTok tok undefined) p
        where keepSnd a b = b
    keepLongest [] = []
    keepLongest as = [List.maximumBy isLonger as]
    isLonger (a, _) (b, _) = compare (length a) (length b)

twoOrMoreSepBy tok p = pThen3 mklist p (pTok tok pUn) (oneOrMoreSepBy tok p)
    where mklist fst _ rest = fst:rest
```

B.2. Núcleo

Los módulos que se presentan a continuación constituyen el núcleo de la implementación.

B.2.1. Máquina

Este módulo implementa la máquina virtual en sí.

```
module Maquina (minitFromStr, traza, showBoxed) where

import qualified List
import qualified AList
import qualified Arbol
import qualified Heap
import Proc
import Ambient
import Parse
import Boxes
```

Un proceso es identificado por un entero

```
type Pid = Integer
```

La máquina virtual tiene una lista de los procesos, indexada por identificador. Cada uno está asociado al ambiente que lo contiene.

```
data Plist = Plist { list :: AList.AList Pid (Proc, Arbol.Aid),
                     lastpid::Pid }
           deriving (Show)
```

La Máquina consta de una cola de procesos en ejecución, un árbol de ambientes, una lista de procesos y un contador que es usado para generar nombres únicos por el operador ν.

```
type Rqueue = [Pid]
data Mach = Mach { r :: Rqueue,
                   t :: Arbol.Arbol (Ambient Pid),
                   p :: Plist,
                   next_name :: Int}


instance Show Mach where
    show m = ashow (pshow (p m)) (t m) Arbol.topid
```

Para inicializar la máquina le damos un proceso a ejecutar. El estado inicial tiene a dicho proceso, un árbol vacío y el contador de nombres en 1.

```
minit proc = Mach { r = [pid], t = tree, p = plist, next_name = 1 }
    where
    tree = top [pid]
    plist = insertproc (proc, Arbol.getlastid tree)
            (Plist AList.new 0)
    pid = lastpid plist
```

La interfaz pública del módulo exporta esta función que inicializa la máquina a partir de una cadena de caracteres, usando el analizador sintáctico del módulo `Parser`.

```
minitFromStr = minit . parseString
```

La Máquina avanza buscando la regla siguiente dado su estado actual y aplicándola a dicho estado La *regla siguiente* se define como la primera aplicable al tipo de proceso que está al inicio de la cola y tal que su condición es verdadera.

```
avanzaMaquina m = (act . siguienteRegla) m m
```

```
siguienteRegla m = head [ r | r <- siguientesReglas, cond r m]
    where
    siguientesReglas = obtenRegla reglas (fst (getproc (p m) firstpid))
    firstpid = head (r m)
```

Una traza de ejecución es la lista de estados de la máquina a partir de un estado inicial. Un estado de la máquina es final cuando la cola de ejecución está vacía.

```
traza m = if null (r m)
          then [(m, "")]
          else (m, nombre (siguienteRegla m)):traza (avanzaMaquina m)
```

Exportamos también esta función, que muestra en forma de cajas un estado de la máquina.

```
showBoxed m = ambientBox m Arbol.topid
```

Reglas Una regla es un nombre, una condición y una acción. El nombre se usa sólo para informar al usuario sobre la regla elegida. La condición es una función booleana sobre los estados, que determina si la regla es aplicable al estado dado. La acción es una función de estados a estados, que determina la transformación dada por esta regla.

```
data Regla a = R { nombre :: String, cond :: a -> Bool, act :: a -> a}

instance Show (Regla a) where
    show = nombre
```

Tenemos la lista de reglas, indexada por tipo de proceso. La Lista es una `AList`, cada llave es una cadena y cada entrada una lista de reglas.

```
true _ = True
false _ = False

andf fs m = and [ f m | f <- fs ]

reglas = AList.listToAList [
             ("Par", [R {nombre = "Par",
                         cond = true,
                         act = sustPar} ]),
             ("Null", [R {nombre = "Null",
```

```
                cond = true,
                act = remNull}]),
("Rest", [R {nombre = "Rest",
             cond = true,
             act = do_rest}]),
 ("Amb", [R {nombre = "NewAmb",
             cond = no_opens,
             act = newamb},
          R {nombre = "OpenNewAmb",
             cond = true,
             act = open_new_amb}]),
 ("Recv", [R {nombre = "RecvToHeap",
              cond=no_writers,
              act=comm_to_heap},
           R {nombre = "Recv",
              cond=true,
              act=do_recv}]),
 ("BangRecv", [R {nombre = "BangRecvToHeap",
                  cond=no_writers,
                  act=comm_to_heap},
                R {nombre = "BangRecv",
                  cond=true,
                  act=do_brecv}]),
 ("Send", [R {nombre = "SendToHeap",
              cond=no_readers,
              act=comm_to_heap},
           R {nombre = "Send",
              cond=heap_has_bang_recv,
              act=do_send_to_bang},
           R {nombre = "Send",
              cond=true,
              act=do_send}]),
 ("Seq", [R {nombre = "DoOpen",
             cond = andf [next_is_open,
                          exists_proper_child],
             act = do_open},
         R {nombre = "OpenToHeap",
            cond = next_is_open,
            act = open_to_heap },
          R {nombre = "DoIn",
             cond = andf [next_is_in,
                          exists_proper_sibbling],
```

```
                act = do_in},
            R {nombre = "InToHeap",
                cond = next_is_in,
                act = in_to_heap},
             R {nombre = "DoOut",
                cond = andf [next_is_out,
                             parent_has_proper_name],
                act = do_out},
            R {nombre = "OutToHeap",
                cond = next_is_out,
                act = out_to_heap}]),
        ("BangMove", [R {nombre = "DoBOpen",
                cond = andf [next_is_bopen,
                             exists_proper_child],
                act = do_bopen},
            R {nombre = "BOpenToHeap",
                cond = next_is_bopen,
                act = open_to_heap },
             R {nombre = "DoBIn",
                cond = andf [next_is_bin,
                             exists_proper_sibbling],
                act = do_bin},
            R {nombre = "BInToHeap",
                cond = next_is_bin,
                act = in_to_heap},
             R {nombre = "DoBOut",
                cond = andf [next_is_bout,
                             parent_has_proper_name],
                act = do_bout},
            R {nombre = "BOutToHeap",
                cond = next_is_bout,
                act = out_to_heap}])
                  ]
```

Dado un proceso, encontramos las reglas que pueden aplicar

```
obtenRegla r Null = AList.getelem r "Null"
obtenRegla r (Rest n p) = AList.getelem r "Rest"
obtenRegla r (Send e) = AList.getelem r "Send"
obtenRegla r (Recv n p) = AList.getelem r "Recv"
obtenRegla r (Amb n p) = AList.getelem r "Amb"
obtenRegla r (Par ps) = AList.getelem r "Par"
obtenRegla r (Seq e p) = AList.getelem r "Seq"
```

```
obtenRegla r (BangMove e p) = AList.getelem r "BangMove"
obtenRegla r (BangRecv e p) = AList.getelem r "BangRecv"
```

Acciones En lo que sigue se definen las condiciones y acciones para cada una de las reglas.

Null Elimina el proceso

```
remNull m = m { r = queue, t = tree, p = plist } where
    pid = head ( r m )
    ambid = snd (getproc (p m) pid)
    queue = tail ( r m )
    tree = substproc (t m) ambid pid []
    plist = deleteproc (p m) pid
```

Par Sustituye el proceso por los componentes

```
sustPar m = m { r = queue, t = tree, p = plist } where
    pid = head ( r m )
    oldp = getproc (p m) pid
    ambid = snd oldp
    procs = let ps (Par qs) = qs in [ (p, ambid) | p <- ps (fst oldp) ]
    newprocs = insertprocs procs (p m)
    pids = [(lastpid (p m) + 1)..(lastpid newprocs)]
     -- La linea anterior expone la implementación
    queue = (tail ( r m )) ++ pids
    tree = substproc (t m) ambid pid pids
    plist = deleteproc newprocs pid
```

Rest Crear un nombre nuevo

```
do_rest m = m { p = plist, next_name = (next_name m) + 1 }
    where
    plist = replaceproc (p m) pid (newproc, aid)
    pid = head (r m)
    (oldproc, aid) = getproc (p m) pid
    new_name = Name ('_':(show (next_name m)))
    old_name = getname oldproc
        where
        getname (Rest (Name a) _) = Name a
```

```
        getname (Rest (Path (Name a:_)) _) = Name a
        getname _ = error "Proceso no es Rest o Exp mal formada en do_rest"
    newproc = subst old_name new_name (next oldproc)
```

Recv El caso en que no hay un Send listo en el Heap lo maneja comm_to_heap.

```
do_recv m = m { t = newtree, p = plist }
    where
    pid = head (r m)
    (oldp, aid) = getproc (p m) pid
    name = getname oldp
        where
        getname (Recv (Name a) _) = Name a
        getname (Recv (Path (Name a:_)) _) = Name a
        getname _ = error "Proceso no es Recv o Exp mal formada en do_recv"
    send_pid = head (Heap.getcomms h "")
    h = heap (Arbol.getdatum (t m) aid)
    (Send (Path [exp]), _) = getproc (p m) send_pid
    plist = replaceproc (p m) pid (subst name exp (next oldp), aid)
    newh = Heap.remfstcomm h ""
    newamb = (Arbol.getdatum (t m) aid) { heap = newh}
    newtree = Arbol.replacedatum (t m) aid newamb

do_brecv m = m { r = queue, t = newtree, p = plist }
    where
    pid = head (r m)
    queue = (r m) ++ [new_pid]
    new_pid = lastpid plist
    (oldp, aid) = getproc (p m) pid
    name = getname oldp
        where
        getname (BangRecv (Name a) _) = Name a
        getname (BangRecv (Path (Name a:_)) _) = Name a
        getname _ = error "Proceso no es BRecv o Exp mal formada en do_brecv"
    send_pid = head (Heap.getcomms h "")
    h = heap (Arbol.getdatum (t m) aid)
    (Send (Path [exp]), _) = getproc (p m) send_pid
    plist = insertproc (subst name exp (next oldp), aid) (p m)
    newh = Heap.remfstcomm h ""
    oldamb = Arbol.getdatum (t m) aid
```

```
    newamb =  oldamb { heap = newh, procs = new_pid:(procs oldamb)}
    newtree = Arbol.replacedatum (t m) aid newamb
```

Amb Crear ambiente.

Caso 1. No hay opens en el Heap. Se crea el ambiente y se reactivan los ins.

```
no_opens m = null (Heap.getopens h name) where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    ambid = snd (getproc (p m) pid)
    h = heap (Arbol.getdatum (t m) ambid)
    name = getname proc where
                getname (Amb (Name n) _) = n
                getname (Amb (Path [Name n]) _ ) = n
                getname _ = error "Proceso exp[P] con exp /= n"

newamb m = m { r = queue, t = newtree, p = plist } where
    pid = head (r m)
    oldp = getproc (p m) pid
    proc = fst oldp
    parentid = snd oldp
    name = getname proc where
                getname (Amb (Name n) _) = n
                getname (Amb (Path [Name n]) _ ) = n
                getname _ = error "Proceso exp[P] con exp /= n"
    newp = (next proc, newambid)
    plist = insertproc newp (p m)
    newpid = lastpid plist
    newtree1 = addchildren (t m) parentid name [newpid]
    newambid = Arbol.getlastid newtree1
    newtree = substproc newtree1 parentid pid []
    queue = tail (r m) ++ [newpid]
```

Caso 2. Hay opens. Se hace la reducción open n.P | n[Q] -> P|Q .

```
open_new_amb m = m { r = queue, t = newtree, p = plist }
    where
    pid = head (r m)
    queue = (r m) ++ [open_continuation_pid]
    newtree = Arbol.replacedatum (t m) ambid
              (oldamb {heap = newheap, procs = newprocs})
```

```
    (nq, ambid) = getproc (p m) pid
    open = fst (getproc (p m) open_continuation_pid)
    oldamb = Arbol.getdatum (t m) ambid
    oldheap = heap oldamb
    newheap = Heap.remfstopen oldheap name
    newprocs = open_continuation_pid:(procs oldamb)
    open_continuation_pid = Heap.headopen oldheap name
    name = fst_name nq
    plist = replaceproc plist' pid (next nq, ambid)
    plist' = replaceproc (p m) open_continuation_pid (next open, ambid)
```

Send Caso 1. No hay lectores en el Heap.

```
no_readers m = or [null comms, is_send (head comms) ]
    where
    pid = head (r m)
    ambid = snd (getproc (p m) pid)
    h = heap (Arbol.getdatum (t m) ambid)
    comms = Heap.getcomms h ""
    is_send = is_send2 . fst  . (getproc (p m))
    is_send2 (Send _) = True
    is_send2 _ = False

comm_to_heap m = m { r = queue, t = newtree, p = plist }
    where
    queue = tail (r m)
    plist = p m
    newtree = substproc newtree1 ambid pid []
    newtree1 = Arbol.replacedatum (t m) ambid newamb
    newamb = oldamb { heap = newheap }
    newheap = Heap.appendcomm (heap oldamb) "" pid
    ambid = snd (getproc (p m) pid)
    pid = head (r m)
    oldamb = Arbol.getdatum (t m) ambid
```

Caso 2. Hay un lector replicado. Se activa una copia.

```
heap_has_bang_recv m = is_bang_recv proc
    where
    pid = head (r m)
    ambid = snd (getproc (p m) pid)
    h = heap (Arbol.getdatum (t m) ambid)
```

```
    comms = Heap.getcomms h ""
    proc = fst (getproc (p m) (head comms))
    is_bang_recv (BangRecv _ _) = True
    is_bang_recv _ = False

do_send_to_bang m = m { r = queue, t = newtree, p = plist }
    where
    queue = (tail (r m)) ++ [continuation_pid]
    continuation_pid = lastpid plist
    plist = insertproc (continuation, ambid) plist'
    plist' = deleteproc (p m) pid
    newtree1 = Arbol.replacedatum (t m) ambid newamb
    newtree = substproc newtree1 ambid pid [continuation_pid]
    continuation = subst name exp (next oldreader)
    newamb = oldamb { heap = newheap }
    reader_pid = head comms
    comms = Heap.getcomms oldheap ""
    newheap = Heap.appendcomm (Heap.remfstcomm oldheap "" ) "" reader_pid
    oldheap = heap oldamb
    ambid = snd (getproc (p m) pid)
    writer = fst (getproc (p m) pid)
    pid = head (r m)
    oldamb = Arbol.getdatum (t m) ambid
    oldreader = fst (getproc (p m) reader_pid)
    exp = let ex (Send e) = e in ex writer
    name = let na (BangRecv n _) = n in na oldreader
```

Caso 3. Hay un lector normal. Se activa.

```
do_send m = m { r = queue, t = newtree, p = plist }
    where
    queue = (tail (r m)) ++ [reader_pid]
    plist = replaceproc (p m) reader_pid (continuation, ambid)
    newtree' = Arbol.replacedatum (t m) ambid newamb
    newtree = substproc newtree' ambid pid [reader_pid]
    continuation = subst name exp (next oldreader)
    newamb = oldamb { heap = Heap.remfstcomm oldheap "" }
    reader_pid = head comms
    comms = Heap.getcomms oldheap ""
    oldheap = heap oldamb
    ambid = snd (getproc (p m) pid)
    writer = fst (getproc (p m) pid)
    pid = head (r m)
```

```
    oldamb = Arbol.getdatum (t m) ambid
    oldreader = fst (getproc (p m) reader_pid)
    Send exp = writer
    name =  na oldreader
        where na (Recv (Path [n]) _) = n
              na (Recv n _) = n
              na _ = error "Recv mal formado en do_send"
```

Recv Caso 1. No hay escritores en el heap

```
no_writers m = or [null comms, is_recv (head comms) ]
    where
    pid = head (r m)
    ambid = snd (getproc (p m) pid)
    h = heap (Arbol.getdatum (t m) ambid)
    comms = Heap.getcomms h ""
    is_recv = is_recv2 . fst  . (getproc (p m))
    is_recv2 (Recv _ _) = True
    is_recv2 (BangRecv _ _) = True
    is_recv2 _ = False


next_is_open m = is_open proc
    where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    is_open (Seq (Open _ ) _) = True
    is_open (Seq (Path (Open _:_)) _) = True
    is_open _ = False
```

Caso 1. El ambiente requerido existe, ejecutamos el open.

```
exists_proper_child m = not (null (getchildrennamed (t m) aid name))
    where
    pid = head (r m)
    (proc, aid) = getproc (p m) pid
    name = fst_name proc


do_open m = m { r = queue, t = newtree, p = plist }
    where
    queue = (r m)
```

```
        ++ procs_from_child_heap
        ++ woken_opens
        ++ woken_outs
        ++ woken_ins
newtree = disolve newtree' child_aid
newtree' = (foldr (.) id [\tr -> Arbol.replacedatum tr aid newamb
                            | (aid, newamb) <- zip
                                                ([parent_aid]
                                                 ++ gch_aids
                                                 ++ sib_aids)
                                                ([newparent]
                                                 ++ new_gchs
                                                 ++ new_sibs)] ) (t m)
plist = replaceproc plist' pid (next proc, aid)
plist' = pmap repl_aid (p m) where
                    repl_aid (p, id)|id == child_aid = (p, aid)
                                    |otherwise       = (p, id)
pid = head (r m)
(proc, aid) = getproc (p m) pid
child_name = fst_name proc
child_aid = head (getchildrennamed (t m) aid child_name)
child = Arbol.getdatum (t m) child_aid
parent_aid = Arbol.getparent (t m) child_aid
parent = Arbol.getdatum (t m) parent_aid
procs_from_child_heap = Heap.allprocs (heap child)
gch_aids = Arbol.getchildren (t m) child_aid
gchs = [Arbol.getdatum (t m) id |  id <- gch_aids ]
gch_names = List.nub [ name gchild | gchild <- gchs ]
sib_aids = Arbol.getchildren (t m) parent_aid
sibs = [Arbol.getdatum (t m) id |  id <- sib_aids ]
newparent = parent { heap = wake_opens (heap parent),
                            procs = (procs parent) ++ woken_opens }
    where
    wake_opens he = (foldr (.) id [\h -> Heap.remfstopen h k
                                       | k <- gch_names] ) he
woken_opens = concat [Heap.getfstopen (heap parent) k | k <- gch_names]
new_gchs = [gch { heap = wake_out (heap gch),
                         procs = (procs gch) ++
                                 Heap.getfstout (heap gch) (name parent) }
            | gch <- gchs ]
    where
    wake_out h = Heap.remfstout h (name parent)
```

```
    woken_outs = concat [Heap.getfstout (heap gch) (name parent)
                              | gch <- gchs]
    new_sibs = [sib { heap = wake_ins (heap sib),
                      procs = (procs sib) ++
                              concat [Heap.getfstin (heap sib) k
                                          | k <- gch_names]}
                | sib <- sibs ]
        where
        wake_ins he =  (foldr (.) id [\h -> Heap.remfstin h k
                                        | k <- gch_names] ) he
    woken_ins = concat [Heap.getfstin (heap sib) k
                              | sib <- sibs, k <- gch_names]
```

Caso 2. El ambiente no existe, subimos al heap.

```
open_to_heap m = m { r = queue, t = newtree}
    where
    pid:queue = (r m)
    newtree = substproc newtree' ambid pid []
    newtree' = Arbol.replacedatum (t m) ambid (oldamb { heap = newheap })
    (proc, ambid) = getproc (p m) pid
    oldamb = Arbol.getdatum (t m) ambid
    newheap = Heap.appendopen (heap oldamb) name pid
    name = fst_name proc
```

```
next_is_in m = is_in proc
    where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    is_in (Seq (In _ ) _) = True
    is_in (Seq (Path (In _:_)) _) = True
    is_in _ = False
```

Caso 1. Existe el ambiente requerido.

```
exists_proper_sibbling m = not (null (getsibblingsnamed (t m) aid name))
    where
    pid = head (r m)
    (proc, aid) = getproc (p m) pid
    name = fst_name proc
```

```
do_in m = m {r = queue, t = newtree, p = plist}
    where
    pid = head (r m)
    queue = (r m) ++ newprocs
    (newtree, newprocs) = go_in (t m) aid name
    plist = replaceproc (p m) pid (next proc, aid)
    (proc, aid) = getproc (p m) pid
    name = fst_name proc
```

Caso 2. No existe el ambiente, subimos al heap.

```
in_to_heap m = m { r = queue, t = newtree}
    where
    pid:queue = (r m)
    newtree = substproc newtree' ambid pid []
    newtree' = Arbol.replacedatum (t m) ambid (oldamb { heap = newheap })
    (proc, ambid) = getproc (p m) pid
    oldamb = Arbol.getdatum (t m) ambid
    newheap = Heap.appendin (heap oldamb) name pid
    name = fst_name proc
```

Función auxiliar para in

Mover un ambiente dentro de un hermano dado. Regresa un par con el nuevo árbol y la lista de procesos que se deben poner en la cola de ejecución.

```
go_in t aid sibname = (newtree, newprocs ++ newsibprocs)
    where
    newtree = Arbol.replacedatum newtree' sibbling newsib
    newtree' = Arbol.replacedatum newtree'' aid newamb
    newtree'' = Arbol.reparent t aid sibbling
    oldamb = Arbol.getdatum t aid
    sibbling = head  (getsibblingsnamed t aid sibname)
    oldsib = Arbol.getdatum t sibbling
    newsib = oldsib { heap = newsibheap,
                      procs = (procs oldsib) ++ newsibprocs }
    newamb = oldamb { heap = newheap, procs = (procs oldamb) ++ newprocs }
    oldheap = heap oldamb
    oldsibheap = heap oldsib
    newheap' = Heap.remfstout oldheap sibname
    newheap = (foldr (.) id [\h -> Heap.remfstin h k |
                     k <- AList.allkeys (Heap.ins newheap')]) newheap'
    newprocs = concat ((Heap.getfstout oldheap sibname):
                [Heap.getfstin oldheap n|
```

```
                n <- AList.allkeys (Heap.ins oldheap)])
    newsibheap = Heap.remfstopen oldsibheap (name oldamb)
    newsibprocs = Heap.getfstopen oldsibheap (name oldamb)



next_is_out m = is_out proc
    where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    is_out (Seq (Out _ ) _) = True
    is_out (Seq (Path (Out _:_)) _) = True
    is_out _ = False
```

Caso 1. Existe el ambiente requerido.

```
parent_has_proper_name m = name == (getparentname (t m) aid)
    where
    pid = head (r m)
    (proc, aid) = getproc (p m) pid
    name = fst_name proc

do_out m = m {r = queue, t = newtree, p = plist}
    where
    pid = head (r m)
    queue = (r m) ++ newprocs
    (newtree, newprocs) = go_out (t m) aid
    plist = replaceproc (p m) pid (next proc, aid)
    (proc, aid) = getproc (p m) pid
    name = fst_name proc
```

Caso 2. No existe el ambiente, subimos al heap.

```
out_to_heap m = m { r = queue, t = newtree}
    where
    pid:queue = (r m)
    newtree = substproc newtree' ambid pid []
    newtree' = Arbol.replacedatum (t m) ambid (oldamb { heap = newheap })
    (proc, ambid) = getproc (p m) pid
    oldamb = Arbol.getdatum (t m) ambid
    newheap = Heap.appendout (heap oldamb) name pid
    name = fst_name proc
```

Función auxiliar para out

Mover un ambiente fuera de su padre. Regresa un par con el nuevo árbol y la lista de procesos que se deben poner en la cola de ejecución.

```
go_out t aid = (newtree, newprocs ++ newgpprocs)
    where
    newtree = Arbol.replacedatum newtree' grandparent newgp
    newtree' = Arbol.replacedatum newtree'' aid newamb
    newtree'' = Arbol.reparent t aid grandparent
    oldamb = Arbol.getdatum t aid
    grandparent = Arbol.getparent t (Arbol.getparent t aid)
    gp_name = getparentname t (Arbol.getparent t aid)
    newamb = oldamb { heap = newheap, procs = (procs oldamb) ++ newprocs }
    gp = Arbol.getdatum t grandparent
    newgp = gp { heap = newgpheap, procs = (procs gp) ++ newgpprocs }
    oldheap = heap oldamb
    newheap' = Heap.remfstout oldheap gp_name
    newheap = (foldr (.) id [\h -> Heap.remfstin h k |
                      k <- AList.allkeys (Heap.ins newheap')]) newheap'
    newprocs = concat ((Heap.getfstout oldheap gp_name):
                [Heap.getfstin oldheap n|
                n <- AList.allkeys (Heap.ins oldheap)])
    oldgpheap = heap gp
    newgpheap = Heap.remfstopen oldgpheap (name oldamb)
    newgpprocs = Heap.getfstopen oldgpheap (name oldamb)



next_is_bopen m = is_open proc
    where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    is_open (BangMove (Open _ ) _) = True
    is_open (BangMove (Path (Open _:_)) _) = True
    is_open _ = False


do_bopen m = m { r = queue, t = newtree, p = plist }
    where
    queue = (r m)
            ++ procs_from_child_heap
            ++ woken_opens
            ++ woken_outs
```

```
        ++ woken_ins
        ++ [new_pid]
newtree = disolve newtree' child_aid
newtree' = (foldr (.) id [\tr -> Arbol.replacedatum tr aid newamb
                            | (aid, newamb) <- zip
                                                ([parent_aid]
                                                 ++ gch_aids
                                                 ++ sib_aids)
                                                ([newparent]
                                                 ++ new_gchs
                                                 ++ new_sibs)] ) (t m)
plist = insertproc (next proc, aid) plist'
plist' = pmap repl_aid (p m) where
                      repl_aid (p, id)|id == child_aid = (p, aid)
                                      |otherwise       = (p, id)
new_pid = lastpid plist
pid = head (r m)
(proc, aid) = getproc (p m) pid
child_name = fst_name proc
child_aid = head (getchildrennamed (t m) aid child_name)
child = Arbol.getdatum (t m) child_aid
parent_aid = Arbol.getparent (t m) child_aid
parent = Arbol.getdatum (t m) parent_aid
procs_from_child_heap = Heap.allprocs (heap child)
gch_aids = Arbol.getchildren (t m) child_aid
gchs = [Arbol.getdatum (t m) id |  id <- gch_aids ]
gch_names = List.nub [ name gchild | gchild <- gchs ]
sib_aids = Arbol.getchildren (t m) parent_aid
sibs = [Arbol.getdatum (t m) id |  id <- sib_aids ]
newparent = parent { heap = wake_opens (heap parent),
                     procs = (procs parent) ++ woken_opens ++ [new_pid] }
    where
    wake_opens he = (foldr (.) id [\h -> Heap.remfstopen h k
                                       | k <- gch_names] ) he
woken_opens = concat [Heap.getfstopen (heap parent) k | k <- gch_names]
new_gchs = [gch { heap = wake_out (heap gch),
                         procs = (procs gch) ++
                                 Heap.getfstout (heap gch) (name parent) }
            | gch <- gchs ]
    where
    wake_out h = Heap.remfstout h (name parent)
woken_outs = concat [Heap.getfstout (heap gch) (name parent)
```

```
                              | gch <- gchs]
    new_sibs = [sib { heap = wake_ins (heap sib),
                      procs = (procs sib) ++
                              concat [Heap.getfstin (heap sib) k
                                      | k <- gch_names]}
                | sib <- sibs ]
        where
        wake_ins he =  (foldr (.) id [\h -> Heap.remfstin h k
                                      | k <- gch_names] ) he
    woken_ins = concat [Heap.getfstin (heap sib) k
                            | sib <- sibs, k <- gch_names]

next_is_bin m = is_in proc
    where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    is_in (BangMove (In _ ) _) = True
    is_in (BangMove (Path (In _:_)) _) = True
    is_in _ = False

do_bin m = m {r = queue, t = newtree, p = plist}
    where
    pid = head (r m)
    queue = (r m) ++ newprocs ++ [new_pid]
    (newtree', newprocs) = go_in (t m) aid name
    newtree = Arbol.replacedatum newtree' aid newamb
    newamb = oldamb { procs = new_pid:(procs oldamb) }
    oldamb = Arbol.getdatum newtree' aid
    plist = insertproc (next proc, aid) (p m)
    new_pid = lastpid plist
    (proc, aid) = getproc (p m) pid
    name = fst_name proc

next_is_bout m = is_out proc
    where
    pid = head (r m)
    proc = fst (getproc (p m) pid)
    is_out (BangMove (Out _ ) _) = True
    is_out (BangMove (Path (Out _:_)) _) = True
    is_out _ = False

do_bout m = m {r = queue, t = newtree, p = plist}
```

```
    where
    pid = head (r m)
    queue = (r m) ++ newprocs ++ [new_pid]
    (newtree', newprocs) = go_out (t m) aid
    newtree = Arbol.replacedatum newtree' aid newamb
    newamb = oldamb { procs = new_pid:(procs oldamb) }
    oldamb = Arbol.getdatum newtree' aid
    plist = insertproc (next proc, aid) (p m)
    new_pid = lastpid plist
    (proc, aid) = getproc (p m) pid
    name = fst_name proc
```

Funciones auxiliares Las funciones de esta última sección son auxiliares al resto de la implementación.

Tenemos funciones para manipular la lista.

```
insertproc p (Plist alist pid)= Plist (AList.insert (newpid, p) alist) newpid
    where
    newpid = pid + 1

insertprocs list = foldr (.) id [insertproc proc | proc <- list]

getproc (Plist alist _) = AList.getelem alist
deleteproc (Plist alist pid) p = Plist (AList.deletekey alist p) pid
replaceproc (Plist alist pid) pi p = Plist (AList.insert (pi, p) alist) pid
pmap f (Plist l pid) = Plist (AList.amap f l) pid

pshow plist = show . fst . (getproc plist)
```

Esta función toma un árbol de ambientes y un identificador y muestra ese ambiente y a sus hijos en forma de caja.

```
ambientBox m aid = Frame (name a) (Sbox ps, Sbox hps, Hbox childs)
    where
    a = Arbol.getdatum (t m) aid
    ps = concat (List.intersperse "|" [ pshow (p m) p' | p' <- procs a ])
    hps = concat (List.intersperse "|" [ pshow (p m) p'
                                       | p' <- Heap.allprocs (heap a)])
    childs = [ ambientBox m child | child <- Arbol.getchildren (t m) aid ]
```

B.2.2. Ambientes

Este módulo implementa la representación de los ambientes y las operaciones sobre la misma.

```
module Ambient (Ambient, name, heap, procs, top, ashow, addchildren,
        getchildrennamed, getsibblingsnamed, getparentname,
        substproc, disolve)
where
import qualified Arbol
import qualified Heap
import qualified List
import qualified AList
```

Un ambiente es una cuarteta de su nombre, sus procesos, sus hijos y un Heap. Implementamos la relación de padre-hijo poniendo los ambientes en un árbol. El ambiente necesita conocer su propia posición en el árbol.

En este módulo definimos ambientes generales que en la máquina se especializan a ambientes de procesos.

```
data Show a => Ambient a = Amb { name :: String,
                                 procs :: [a],
                                 aid :: Arbol.Aid,
                                 heap :: Heap.Heap String a }
                         deriving (Show)
```

Para mostrar un ambiente mostramos su nombre, después, entre corchetes y separados por '|', cada elemento de PROCS y luego, recursivamente, a sus hijos.

```
ashow pshow t aid = if name amb == ""
                    then
                    scontent
                    else
                    name amb ++ "["  ++ scontent ++ "]"
    where
    amb = Arbol.getdatum t aid
    children = Arbol.getchildren t aid
    scontent = concat ( sprocs ++ ["␣|␣("] ++ sheap ++ ["␣)␣|␣"] ++ childs )
    sprocs = List.intersperse "|" [pshow p| p <- procs amb]
    sheap = List.intersperse "|" [pshow p | p <-
                                 (allins ++ allouts ++ allopens ++ allcomms )]
    childs = List.intersperse "|" [ashow pshow t a | a <- children ]
```

```
    allins = concat [ Heap.getins (heap amb) key
                        | key <- AList.allkeys (Heap.ins (heap amb))]
    allouts = concat [ Heap.getouts (heap amb) key
                         | key <- AList.allkeys (Heap.outs (heap amb))]
    allopens = concat [ Heap.getopens (heap amb) key
                          | key <- AList.allkeys (Heap.opens (heap amb))]
    allcomms = concat [ Heap.getcomms (heap amb) key
                          | key <- AList.allkeys (Heap.comms (heap amb))]
```

Los ambientes son identificados por su identificador en el árbol.

```
instance Show a => Eq (Ambient a) where
  x == y   = (aid x) == (aid y)
```

Cada árbol comienza en un ambiente 'Top' que contiene a los demás.

```
top ps = arbol where
    arbol = Arbol.nuevo
            (Amb {name = "",
                  procs = ps,
                  aid = Arbol.getlastid arbol,
                  heap = Heap.empty})
```

Algunas primitivas de manipulación:

Añadir un hijo de nombre NEWNAME al ambiente AID

```
addchildren t aid newname ps = newtree
    where
    newtree = Arbol.addchildren t aid
              (Amb { name = newname,
                     procs = ps,
                     aid = Arbol.getlastid newtree,
                     heap = Heap.empty })
```

Obtener los hijos de nombre GIVEN

```
getchildrennamed t aid given = [ aids | aids <- (Arbol.getchildren t aid),
                                 name (Arbol.getdatum t aids) == given ]
```

Obtener los hermanos de nombre GIVEN

```
getsibblingsnamed t aid given = [ aids |
                                  aids <- (Arbol.getchildren t
                                           (Arbol.getparent t aid)),
                                  aids /= aid,
                                  name (Arbol.getdatum t aids) == given ]
```

Obtener el nombre del padre

```
getparentname t = name . (Arbol.getdatum t) . (Arbol.getparent t)
```

Sustituir un proceso por otros

```
substproc t aid pid pids = Arbol.replacedatum t aid newamb where
    amb = (Arbol.getdatum t aid)
    newamb = amb { procs = ( [ p | p <- (procs amb), p /= pid ] ++ pids )}
```

Disolver un ambiente en su padre

```
disolve t child_aid = Arbol.deletechildren newtree parent_aid child_aid
    where
    child = Arbol.getdatum t child_aid
    parent_aid = Arbol.getparent t child_aid
    parent = Arbol.getdatum t parent_aid
    newparent = parent { procs = procs parent ++
                                 procs child ++ procs_from_child_heap }
    old_procs = procs parent
    procs_from_child_heap = Heap.allprocs (heap child)
    newtree = Arbol.replacedatum t parent_aid newparent
```

B.2.3. Procesos

```
-- Tipos de datos para la máquina virtual.
module Proc where

import List

data Expr = Nil
          | Name String
          | In Expr
          | Out Expr
          | Open Expr
          | Path [Expr]

instance Show Expr where
    show Nil = []
    show (Name n) = n
    show (In e) = "in " ++ show e
    show (Out e) = "out " ++ show e
    show (Open e) = "open " ++ show e
```

```
    show (Path es) = concat (intersperse "." [ show e | e <- es ])


data Proc = Null
          | Rest Expr Proc
          | Send Expr
          | Recv Expr Proc
          | Amb Expr Proc
          | Par [Proc]
          | Seq Expr Proc
          | BangMove Expr Proc -- !in n.P, !out n.P, !open n.P
          | BangRecv Expr Proc -- !(n).P


instance Show Proc where
    show Null = "0"
    show (Rest n p) = concat [ "(nu ", show n, ")", show p]
    show (Send e) = concat [ "<", show e, ">" ]
    show (Recv n p) = concat [ "(", show n, ").", show p]
    show (Amb (Name "_") p) = show p
    show (Amb n p) = concat [ show n, "[", show p, "]" ]
    show (Par ps) = concat ( ["{"]
                           ++ (intersperse "||" [show p | p <- ps ])
                           ++ ["}"] )
    show (Seq e p) = concat [show e, ".", show p]
    show (BangMove e p) = concat [ "!", show (Seq e p) ]
    show (BangRecv e p) = concat [ "!", show (Recv e p) ]

-- Sustituciones en expresiones
e_subst _ _ Nil = Nil
e_subst (Name n) m (Name l) = if n == l
                              then m
                              else Name l
e_subst e f (In g) = In (e_subst e f g)
e_subst e f (Out g) = Out (e_subst e f g)
e_subst e f (Open g) = Open (e_subst e f g)
e_subst e f (Path gs) = Path [e_subst e f g| g <- gs]
e_subst _ _ _ = error "Sustitución en expresión no definida"

-- Sustituciones en procesos
subst _ _ Null = Null
subst e f (Send g) = Send (e_subst e f g)
```

```
subst e f (Seq g p) = Seq (e_subst e f g) (subst e f p)
subst e f (Amb g p) = Amb (e_subst e f g) (subst e f p)
subst e f (BangMove g p) = (BangMove (e_subst e f g) (subst e f p))
subst e f (Par ps) = Par [ subst e f p | p <- ps ]
subst (Name n) (Name m) (Rest (Name l) p) = if n == l
                                            then (Rest (Name l) p)
                                            else (Rest (Name l) (subst
                                                                 (Name n)
                                                                 (Name m)
                                                                 p))
subst (Name n) (Path [Name m]) (Rest (Name l) p) = if n == l
                                                   then (Rest (Name l) p)
                                                   else (Rest (Name l) (subst
                                                                        (Name n)
                                                                        (Name m)
                                                                        p))
subst (Name n) (Name m) (Recv (Name l) p) = if n == l
                                            then (Recv (Name l) p)
                                            else (Recv (Name l) (subst
                                                                 (Name n)
                                                                 (Name m)
                                                                 p))
subst (Name n) (Name m) (BangRecv (Name l) p) = if n == l
                                                then (BangRecv (Name l) p)
                                                else (BangRecv
                                                      (Name l)
                                                      (subst
                                                       (Name n)
                                                       (Name m)
                                                       p))
subst e f p = error ("Sustitución␣en␣proceso␣no␣definida:␣"
                     ++ show e ++ "␣"
                     ++ show f ++ "␣"
                     ++ show p)

-- Continuación de un proceso
next Null = error "No␣continuation␣for␣Null"
next (Rest e p) = p
next (Send e) = error "No␣continuation␣for␣send"
next (Recv e p) = p
next (Amb e p) = p
next (Par ps) = Par ps
```

```
next (Seq (Path [e]) p) = p
next (Seq (Path (e:es)) p) = Seq (Path es) p
next (Seq e p) = p
next (BangMove (Path [e]) p) = p
next (BangMove (Path (e:es)) p) = Seq (Path es) p
next (BangRecv e p) = p

fst_name (Seq (Path (Open (Name n):_)) _) = n
fst_name (Seq (Path (Out (Name n):_)) _) = n
fst_name (Seq (Path (In (Name n):_)) _) = n
fst_name (Seq (Open (Name n)) _) = n
fst_name (Seq (Out (Name n)) _) = n
fst_name (Seq (In (Name n)) _) = n
fst_name (BangMove (Path (Open (Name n):_)) _) = n
fst_name (BangMove (Path (Out (Name n):_)) _) = n
fst_name (BangMove (Path (In (Name n):_)) _) = n
fst_name (BangMove (Open (Name n)) _) = n
fst_name (BangMove (Out (Name n)) _) = n
fst_name (BangMove (In (Name n)) _) = n
fst_name (Amb (Name n) _) = n
fst_name _ = error "Intento␣de␣obtener␣el␣primer␣nombre␣de␣un␣proceso␣erroneo"

------- Ejemplos y pruebas -------
proc_ejemplo = Par [Amb (Name "l") Null,
                    Send (Path [In (Name "n"),
                                Out (Name "m")]) ,
                    Recv (Name "n")
                    (Amb (Name "m") (Seq (Name "n") Null))]
```

B.2.4. Árboles

```
module Arbol(Arbol, Aid, nuevo, getfromid, getlastid, addchildren,
             getdatum, replacedatum, getchildren, getsibblings,
             getparent, deletechildren, reparent, topid )
where

import qualified AList
import qualified List

newtype Aid = Aid Integer
    deriving (Eq, Ord, Show)
```

```
instance Num Aid where
    fromInteger n = Aid n

data Eq a => Node a = Node { parent   :: Aid,
                             self     :: Aid,
                             datum    :: a,
                             children :: [Aid] }
                      deriving (Eq, Show)

data Arbol a = Arbol { alist :: (AList.AList Aid (Node a)),
                       lastkey :: Aid }
               deriving (Show)

incaid (Aid n) = Aid (n+1)

topid = Aid 0

nuevo a = Arbol (AList.insert
                 (topid ,
                  (Node { parent = (Aid 0),
                          self = (Aid 0),
                          datum = a,
                          children = [] }))
                 AList.new)
          topid

getlastid (Arbol _ aid) = aid

getfromid (Arbol alist _) key = AList.getelem alist key

prueba = nuevo 0
nodo = getfromid prueba (Aid 0)

getdatum tree key = datum (getfromid tree key)

getchildren tree = children . getfromid tree

getparent tree = parent . getfromid tree

getsibblings tree = getchildren tree . getparent tree

addchildren (Arbol alist lastkey) key a =
```

```
    Arbol ( AList.insert
            ( newkey,
              (Node { parent = key,
                      self = newkey,
                      datum = a,
                      children = [] }))
            newlist) newkey
    where
    newkey = incaid lastkey
    newlist = AList.insert (key, newparent) alist
    oldparent = getfromid (Arbol alist lastkey) key
    oldchildren = getchildren
                  (Arbol alist lastkey)
                  (self oldparent)
    newchildren = oldchildren ++ [newkey]
    newparent = oldparent {children = newchildren}


replacedatum (Arbol alist lastkey) key a =
    Arbol ( AList.insert (key, newnode) alist) lastkey
    where
    oldnode = getfromid (Arbol alist lastkey) key
    newnode = oldnode { datum = a }

deletechildren (Arbol alist lastkey) parent child =
    Arbol newalist lastkey
    where
    oldparent = getfromid (Arbol alist lastkey) parent
    newchildren = List.delete child (children oldparent)
    newparent = oldparent { children = newchildren }
    newalist' = AList.insert (parent, newparent)
                (AList.deletekey alist child)
    Arbol newalist _= (foldr
                       (.)
                       id
                       [\t -> reparent t gch parent
                            | gch <- getchildren (Arbol alist lastkey) child ])
                      (Arbol newalist' lastkey)

reparent t child_id newparent_id = t { alist = newalist }
    where
    oldparent_id = parent oldchild
```

```
    oldchild = getfromid t child_id
    newparent = getfromid t newparent_id
    oldparent = getfromid t oldparent_id
    newchild = oldchild { parent = newparent_id }
    newoldparent = oldparent { children = [c | c <- children oldparent,
                                            c /= child_id ] }
    newnewparent = newparent { children = child_id:(children newparent)}
    newalist = AList.insert (child_id, newchild) newalist'
    newalist' = AList.insert (newparent_id, newnewparent) newalist''
    newalist'' = AList.insert (oldparent_id, newoldparent) (alist t)
```

B.3. Bibliotecas de apoyo

Los módulos restantes implementan rutinas de apoyo para el núcleo, por ejemplo estructuras de datos.

B.3.1. Listas

```
module AList(AList, new, getelem, getelemdef, insert,
             showkeys, deletekey, haskey, allkeys,
             listToAList, amap)
where

newtype Eq a => AList a b = AList [ (a, b) ]

instance (Show a, Show b, Eq a) => Show (AList a b) where
    show (AList pairs) = show ( toSList pairs )
        where
        toSList [] = []
        toSList ((k,d):ps) = (concat [show k, " -> ", show d]):(toSList ps)

amap f (AList l) = AList [ (a, f b) | (a, b) <- l ]

new :: Eq a => AList a b
new = AList []

listToAList rs = AList rs

getelem (AList []) key = error ("Key not in list: " ++ (show key))
getelem (AList ((a,b):as)) key = if key == a
                                 then b
```

```
                                  else getelem (AList as) key

getelemdef def alist key = if haskey alist key
                           then getelem alist key
                           else def

insert (a,b) (AList []) = AList [ (a,b) ]
insert (a,b) (AList ((c,d):cs)) = if a == c
                                  then prepend (a,b) (AList cs)
                                  else prepend (c,d) (insert (a,b) (AList cs))
    where
    prepend a (AList (as)) = AList (a:as)

showkeys (AList as) = foldl insertspace "" [(show . fst) a | a <- as ]
    where
    insertspace a b = a ++ "␣" ++ b

deletekey (AList as) key = AList [ (k, a) | (k, a) <- as , k/= key ]

haskey (AList as) key = or [ k == key | (k, a) <- as ]

allkeys (AList as) = [fst pair | pair <- as]
```

B.3.2. Heaps

```
module Heap where

import qualified AList

data Eq a=> Heap a b = Heap { ins, outs, opens, comms :: AList.AList a [b] }
                       deriving (Show)

empty :: Eq a => Heap a b
empty = Heap { ins   = AList.new,
               outs  = AList.new,
               opens = AList.new,
               comms = AList.new }

getins   h = ((AList.getelemdef []) . ins) h
getouts  h = ((AList.getelemdef []) . outs) h
getopens h = ((AList.getelemdef []) . opens) h
getcomms h = ((AList.getelemdef []) . comms) h
```

```
headin   h a = head (getins   h a)
headout  h a = head (getouts  h a)
headopen h a = head (getopens h a)
headcomm h a = head (getcomms h a)

myhead [] = []
myhead (a:as) = [a]

getfstin   h a = myhead (getins   h a)
getfstout  h a = myhead (getouts  h a)
getfstopen h a = myhead (getopens h a)
getfstcomm h a = myhead (getcomms h a)

inhaskey   h = (AList.haskey . ins) h
outhaskey  h = (AList.haskey . outs) h
openhaskey h = (AList.haskey . opens) h
commhaskey h = (AList.haskey . comms) h

mytail [] = []
mytail (a:as) = as

remfstin h a = h { ins = AList.insert (a, ps) oldins }
    where
    oldins = ins h
    ps = if inhaskey h a
         then mytail (getins h a)
         else []

remfstout h a = h { outs = AList.insert (a, ps) oldouts }
    where
    oldouts = outs h
    ps = if outhaskey h a
         then mytail (getouts h a)
         else []

remfstopen h a = h { opens = AList.insert (a, ps) oldopens }
    where
    oldopens = opens h
    ps = if openhaskey h a
         then mytail (getopens h a)
         else []
```

```
remfstcomm h a = h { comms = AList.insert (a, ps) oldcomms }
    where
    oldcomms = comms h
    ps = if commhaskey h a
         then mytail (getcomms h a)
         else []

appendin h k b = h { ins = AList.insert (k, bs) oldins }
    where
    oldins = ins h
    bs = if inhaskey h k
         then (getins h k) ++ [b]
         else [b]

appendout h k b = h { outs = AList.insert (k, bs) oldouts }
    where
    oldouts = outs h
    bs = if inhaskey h k
         then (getouts h k) ++ [b]
         else [b]

appendopen h k b = h { opens = AList.insert (k, bs) oldopens }
    where
    oldopens = opens h
    bs = if inhaskey h k
         then (getopens h k) ++ [b]
         else [b]

appendcomm h k b = h { comms = AList.insert (k, bs) oldcomms }
    where
    oldcomms = comms h
    bs = if inhaskey h k
         then (getcomms h k) ++ [b]
         else [b]

preppendin h k b = h { ins = AList.insert (k, bs) oldins }
    where
    oldins = ins h
    bs = if inhaskey h k
         then [b] ++ (getins h k)
         else [b]
```

```
preppendout h k b = h { outs = AList.insert (k, bs) oldouts }
    where
    oldouts = outs h
    bs = if inhaskey h k
         then [b] ++ (getouts h k)
         else [b]

preppendopen h k b = h { opens = AList.insert (k, bs) oldopens }
    where
    oldopens = opens h
    bs = if inhaskey h k
         then [b] ++ (getopens h k)
         else [b]

preppendcomm h k b = h { comms = AList.insert (k, bs) oldcomms }
    where
    oldcomms = comms h
    bs = if inhaskey h k
         then [b] ++ (getcomms h k)
         else [b]

allins h = concatMap (AList.getelem (ins h)) (AList.allkeys (ins h))
allouts h = concatMap (AList.getelem (outs h)) (AList.allkeys (outs h))
allopens h = concatMap (AList.getelem (opens h)) (AList.allkeys (opens h))
allcomms h = concatMap (AList.getelem (comms h)) (AList.allkeys (comms h))

allprocs h = allins h ++ allouts h ++ allopens h ++ allcomms h
```

B.3.3. Boxes

Toma un árbol y lo imprime como una serie de cajas anidadas.

Este módulo es completamente independiente y tiene como único propósito generar la representación en "cajas" que se usa para los ejemplos del apéndice A.

```
module Boxes where

import Arbol
import List

data Box = Hbox [Box]
```

```
    | Sbox String
    | Frame String (Box, Box, Box)

instance Show Box where
    show b = unlines [[b !: (i,j)| j <- [0 .. (width b) - 1] ]
                          | i <- [ 0 .. (height b) - 1] ]

height (Sbox _) = 1
height (Frame _ (a,b,c)) = 2 + height a + height b + height c
height (Hbox bs) = foldr max 0 [ height b | b <- bs ]

width (Sbox a) = length a
width (Frame n (a,b,c)) = (maximum (length n:width a:width b:width c:[])) + 2
width (Hbox bs) = foldr (+) 0 [width b | b <- bs]

(Sbox a) !: (0, n) = a!!n
(Sbox _) !: (_, _) = error "Sbox solo tiene un renglón"

b@(Hbox bs) !: (i, j) | i >= height b = error "i fuera de rango en Hbox"
                      | j >= width b  = error "j fuera de rango en Hbox"
                      | i >= 0 && j >=0  = if i < height box
                                           then box !: (i, newj)
                                           else ' '
                      | otherwise = error "Índice negativo en Hbox"
    where
    (before, after) = span isShorter (List.inits bs)
        where isShorter init = (foldr (+) 0 (map width init)) <= j
    box = last (head after)
    uptoBox = ((foldr (+) 0) . (map width)) (last before)
    newj = j - uptoBox

f@(Frame n (a,b,c)) !: (i, j) | i >= height f = error
                                    "i fuera de rango en Frame"
                              | j >= width f  = error
                                    "j fuera de rango en Frame"
                              | i < 0 || j < 0 = error
                                    "Índice Negativo en Frame"
                              | j == 0 || j == (width f) - 1 = '|'
                              | i == (height f) - 1    = '_'
                              | i == 0 && j <= length n = n !! (j - 1)
                              | i == 0                  = '_'
                              | i == 1 && j <= width a = a !: (0, j - 1)
```

```
    | i == 1                    = ' '
    | i == 2 && j <= width b = b !: (0, j - 1)
    | i == 2                    = ' '
    | i >= 3 && j <= width c = c !: (i - 3, j - 1)
    | i >= 3                    = ' '
```

B.4. Interfaz con la línea de comandos

Los módulos `boxed` y `cli` solo preparan las rutinas para ejecutarlas desde la linea de comandos.

B.4.1. boxed

Ejecuta la expresión de ambientes que se le pasa como parametro, imprimiendo cada paso de la traza en una representación de cajas anidadas.

```
module Main where
import Maquina
import System

nsteps n s = foldr1 (>>) (take n [print m | m <- traza (minitFromStr s)])

reduce s = foldr1 (>>) [ do print (showBoxed m)
                            print ""
                            print regla
                            print ""
                         | (m, regla) <- traza (minitFromStr s)]

main = do
       arg:_ <- getArgs
       reduce arg
```

B.4.2. cli

Ejecuta la expresión de ambientes que se le pasa como parametro, imprimiendo cada paso de la traza.

```
module Main where
import Maquina
```

```
import System

nsteps n s = foldr1 (>>) (take n [print m | m <- traza (minitFromStr s)])

reduce s = foldr1 (>>) [print m | m <- traza (minitFromStr s)]

main = do
       arg:_ <- getArgs
       reduce arg
```

Apéndice C

Manual de usuario

El código fuente de la máquina está disponible en http://www.nul-unu.com/quien/rodrigo/MAC, en forma de un archivo comprimido. Dicho archivo contiene, además del código fuente, dos binarios compilados para Linux.

La implementación de la máquina fue probada con el intérprete Hugs y con el compilador GHC 6.2, ambos sobre plataforma Linux. Distribuciones de ambos, para diversas plataformas, se pueden conseguir desde la página web de la comunidad de Haskell, http://www.haskell.org.

Los archivos `cli.lhs` y `boxed.lhs` contienen la interfaz del programa con el entorno de línea de comandos.

El primero implementa una interfaz sencilla que evalúa una expresión e imprime cada paso de evaluación y la regla usada en una sintaxis similar a la usada como entrada.

El segundo imprime los resultados en un formato de cajas, como en los ejemplos del apéndice A.

En el caso de expresiones cuya evaluación no termina, ambos programas continuarán imprimiendo pasos de reducción indefinidamente. Es responsabilidad del usuario detenerlos en este caso.

Ambos toman su entrada como un argumento en la línea de comandos. Puesto que la expresión puede contener espacios y otros caracteres especiales es necesario entrecomillarla en la forma requerida por el intérprete de comandos que se esté usando.

La sintaxis de entrada es la definida en la sección 3.1. En general difiere de la usada por Gordon y Cardelli en que no está permitido omitir los procesos 0, y en que es necesario encerrar en paréntesis las continuaciones de una capacidad. Por ejemplo, el proceso que ellos escribirían open n en esta sintaxis se debe escribir open $n.(0)$. En general sea liberal colocando paréntesis. Desafortunadamente el analizador sintáctico de la máquina no tiene un buen manejo de errores, por lo que ante una expresión inválida sólo responde "Error de sintaxis".

Hugs Para ejecutar los programas con Hugs, use el *script* `runhugs` proporcionado con la distribución. El formato general es

```
runhugs módulo 'expresión a evaluar'
```

En donde `módulo` es `cli.lhs` o `boxed.lhs`.

GHC El uso del compilador GHC proporciona una velocidad de ejecución un orden de magnitud mayor que la del intérprete Hugs. Para compilar los programas use el comando

```
ghc --make módulo -o ejecutable
```

En donde `módulo` es `cli.lhs` o `boxed.lhs`, como antes, y `ejecutable` es el nombre que se desee dar al ejecutable compilado. Recomendamos usar `cli` y `boxed`, respectivamente.

Una vez compilados, los programas se ejecutan como cualquier otro programa nativo de la plataforma.

```
ejecutable 'expresión a evaluar'
```

En el caso de Unix o semejantes recuerde que el directorio donde se encuentra el ejecutable probablemente no está incluido en la variable `PATH`, por lo que es necesario especificar la ruta explícitamente:

```
./ejecutable 'expresión a evaluar'
```

Bibliografía

[1] Agha, Gul. *A model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] Armstrong, Joe, editor. *Concurrent programming in Erlang.* Prentice Hall, segunda ed$^{\underline{ón}}$., ene. 1996.

[3] Berendregt, Henk P. *The Lambda Calculus.* North Holland, segunda ed$^{\underline{ón}}$., oct. 1984.

[4] Berry, Gérard y Boudol, Gérard. 'The chemical abstract machine.' *Theoretical Computer Science*, 96:págs. 217–248, 1992.

[5] Cardelli, Luca. 'Obliq: A language with distributed scope.' Inf. Téc. SRC-RR-122, Digital System Research Center, jun. 1994. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-122.html.

[6] Cardelli, Luca. 'A Language with Distributed Scope.' En *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, págs. 286–297. New York, NY, 1995.

[7] Cardelli, Luca. 'Abstractions for Mobile Computation.' En *Secure Internet Programming*, págs. 51–94. 1999.

[8] Cardelli, Luca, Donahue, James, Glassman, Lucille, Jordan, Mick, Kalsow, Bill y Nelson, Greg. 'Modula-3 report.' Inf. Téc. SRC-RR-52, Digital System Research Center, sep.

1989. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-052.html.

[9] Cardelli, Luca, Donahue, James, Glassman, Lucille, Jordan, Mick, Kalsow, Bill y Nelson, Greg. 'Modula-3 Language Definition.' *ACM SIGPLAN Notices*, 27, nº 8:págs. 15–42, ago. 1992.

[10] Cardelli, Luca, Ghelli, Giorgio y Gordon, Andrew D. 'Mobility Types for Mobile Ambients.' En Jiří Wiederman, Peter van Emde Boas y Mogens Nielsen, editores, *Proceedings of ICALP '99*, tomo 1644 de *LNCS*, págs. 230–239. Springer, jul. 1999.

[11] Cardelli, Luca, Ghelli, Giorgio y Gordon, Andrew D. 'Ambient Groups and Mobility Types.' En J. van Leeuwen, O. Watanabe, M. Hagiya, P.D. Mosses y T. Ito, editores, *Proceedings of TCS 2000*, tomo 1872 de *LNCS*, págs. 333–347. IFIP, Springer, ago. 2000.

[12] Cardelli, Luca y Gordon, Andrew D. 'Mobile Ambients.' En *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin, Germany, 1998.

[13] Cardelli, Luca y Gordon, Andrew D. 'Types for Mobile Ambients.' En *Proceedings of POPL '99*, págs. 79–92. ACM, 1999.

[14] Chandy, K. Mani. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. ISBN 0-201-05866-9.

[15] Church, Alonzo. *The calculi of lambda conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[16] Ciancarini, P. 'Distributed Programming with Logic Tuple Spaces.' *New Generation Computing*, 12, nº 3:págs. 251–284, 1994.

[17] Conchon, Sylvain y Fessant, Frabice Le. 'Jocaml: Mobile Agents for Objective-Caml.' En *First Internationa Symposium on Agent Systems and Applications (ASA'99)/Third Internationa Symposium on Mobile Agents (MA'99)*. Palm Springs, CA, USA, 1999.

[18] 'Open source Erlang web site.' `http://www.erlang.org/`.

[19] Fournet, Cédric y Gonthier, Georges. 'The Reflexive CHAM and the Join Calculus.' En *Proceedings of the 23rd ACM Symposium on Principles of programming Languages*, págs. 372–385. ACM Press, 1996.

[20] Hoare, C. A. R., editor. *Occam 2 reference manual*. Prentice Hall, nov. 1988.

[21] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, 1995.

[22] Inmos Limited. *Transputer Reference Manual*. Prentice Hall, jul. 1988.

[23] Jones, Geraint y Goldsmith, Michael. *Programming in Occam 2*. Web Edition, 1988, 2001. `http://web.comlab.ox.ac.uk/oucl/work/geraint.jones/publications/book/Pio2/` .

[24] Jones, Simon Peyton, editor. *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press, april 2003. `http://www.haskell.org/onlinereport/`.

[25] Levi, Francesca y Sangiorgi, Davide. 'Controlling Interference in Ambients.' En *Proceedings of POPL '00*, págs. 352–364. ACM, ene. 2000.

[26] Milner, Robin. *Communication and Concurrency*. Prentice Hall, 1989.

[27] Milner, Robin. 'Functions as Processes.' *Journal of Mathematical Structures in Computer Science*, 2, nº 2:págs. 119–141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[28] Milner, Robin, Parrow, Joachim y Walker, David. 'A Calculus of Mobile Processes, Part I.' Inf. Téc. ECS-LFCS-89-85, Laboratory for Foundations of Computer Science, Computer Science Department, Edimburg University, 1989.

[29] Pericas-Geertsen, Santiago M. *XML-Fluent Mobile Ambients*. Tesis Doctoral, Boston University, 2001.

[30] Pous, Damien. 'Ambient Programming in Icobjs.', Jul 2002. `http://www-sop.inria.fr/mimosa/ambicobjs/`.

[31] Reppy, John H. 'CML: A higher order concurrent language.' *ACM SIGPLAN Notices*, 26, nº 6:págs. 293–305, jun. 1991.

[32] Revesz, G. *Lambda-Calculus, combinators, and functional programming*. Cambridge University Press, Cambridge, UK, 1988.

[33] Roman, Gruia-Catalin, McCann, Peter J. y Plun, Jerome Y. 'Mobile UNITY: reasoning and specification in mobile computing.' *ACM Transactions on Software Engineering and Methodology*, 6, nº 3:págs. 250–282, 1997.

[34] Satyanarayanan, M. 'Fundamental Challenges in Mobile Computing.' En *Fifteenth ACM Symposium on Principles of Distributed Computing*. Philadelphia, PA, mayo 1996. http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html.

[35] Sipser, Michael. *Introduction to the theory of computation*. Brooks Cole, primera edón., dic. 1996.

[36] Turing, Alan. 'On Computable Numbers, With an Application to the Entscheidungsproblem.' En *Proceedings of the London Mathematical Society*, tomo 42 de *2*, págs. 230–265. 1936. reprinted in M. David (ed.), The Undecidable, Hewlett, NY: Raven Press, 1965.

[37] Turner, David N. *The polymorphic Pi-calculus: Theory and implementation*. Tesis Doctoral, University of Edinburgh, 1995.

[38] Zimmer, Pascal. 'Subtyping and Typing Algorithms for Mobile Ambients.' En Jerzy Tiuryn, editor, *Proceedings of FoSSaCS 2000*, tomo 1784 de *LNCS*, págs. 375–390. Springer, 2000.

Índice alfabético