Maude como marco semántico ejecutable

UCM

UNIVERSIDAD
COMPLUTENSE
MADRID

TESIS DOCTORAL

José Alberto Verdejo López

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Enero 2003

Maude como marco semántico ejecutable

Memoria presentada para obtener el grado de
Doctor en Informática

José Alberto Verdejo López

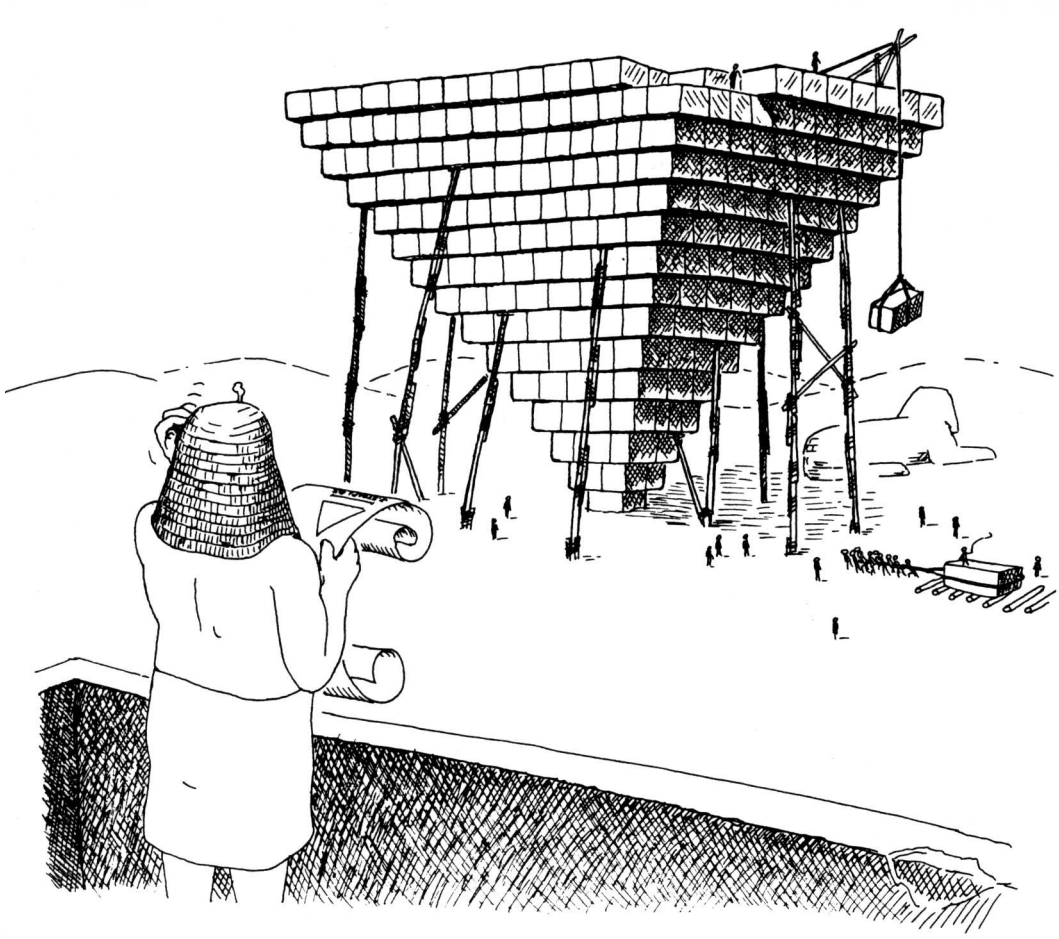
Dirigida por el profesor

Narciso Martí Oliet

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Enero 2003

A mis padres y hermana



No siempre las ejecuciones salen bien, sobre todo si la especificación es ambigua o alguien la interpreta mal.

Resumen

La *lógica de reescritura*, propuesta por José Meseguer en 1990 como marco de unificación de modelos de computación concurrente, es una lógica para razonar sobre sistemas concurrentes con estado que evolucionan por medio de transiciones. Desde su definición, se ha propuesto a la lógica de reescritura como *marco lógico y semántico* en el cual poder expresar de forma natural otras muchas lógicas, lenguajes y modelos de computación. Además, la lógica de reescritura es ejecutable utilizando el lenguaje multiparadigma Maude cuyos módulos son teorías en la lógica de reescritura.

El objetivo principal de esta tesis es extender la idea de la lógica de reescritura y Maude como marco semántico a la idea de *marco semántico ejecutable*. Este objetivo se ha abordado desde diferentes puntos de vista.

En primer lugar, presentamos representaciones ejecutables de semánticas operacionales estructurales. En concreto, hemos estudiado dos implementaciones diferentes de la semántica del cálculo CCS de Milner y su utilización para implementar la lógica modal de Hennessy-Milner; hemos realizado una implementación de una semántica simbólica para el álgebra de procesos LOTOS incluyendo especificaciones de tipos de datos en ACT ONE que son traducidos a módulos Maude, y de una herramienta que permite al usuario ejecutar directamente sus especificaciones LOTOS; y hemos utilizado las mismas técnicas para implementar otros tipos de semánticas operacionales de lenguajes funcionales e imperativos, incluyendo tanto semánticas de evaluación (o paso largo) como semánticas de computación (o paso corto).

En segundo lugar, hemos querido contribuir al desarrollo de una metodología propuesta recientemente por Denker, Meseguer y Talcott para la especificación y análisis de sistemas basada en una jerarquía de métodos incrementalmente más potentes, especificando y analizando tres descripciones ejecutables, a diferentes niveles de abstracción, del protocolo de elección de líder dentro de la especificación del bus multimedia en serie IEEE 1394 (conocido como "FireWire"). En dos de estas descripciones hacemos especial énfasis en los aspectos relacionados con el tiempo, esenciales para este protocolo.

Por último, hemos abordado la dotación de semántica formal a lenguajes de la web semántica, mediante la traducción del lenguaje de descripción de recursos web RDF (*Resource Description Framework*) a Maude. La traducción de documentos RDF a módulos orientados a objetos en Maude ha sido implementada en el propio Maude, gracias a las capacidades de metaprogramación que este ofrece. También hemos mostrado cómo integrar los documentos RDF traducidos con agentes móviles, descritos utilizando Mobile Maude, una extensión de Maude para permitir cómputos móviles.

Agradecimientos

En primer lugar deseo expresar mi agradecimiento a Narciso Martí Oliet, por haber dirigido este trabajo y haberme ayudado en muchos otros, por su continua disponibilidad y apoyo, por toda su ayuda como tutor desde que comencé los estudios de doctorado, y por haber sabido decir muchas veces lo que yo necesitaba oír. Nada de lo que voy a contar a continuación habría sido posible sin él. Gracias, Narciso.

También quiero dar las gracias a David de Frutos, por haberme introducido en el mundo de los métodos formales, por haber contestado siempre mis dudas de informático, y por su trabajo de revisión de esta tesis.

Dentro del Departamento de Sistemas Informáticos y Programación siempre me he sentido muy a gusto, apoyado y ayudado. Quiero agradecer a Luis Llana toda su ayuda, todo el tiempo que dedicamos juntos al estudio de la semántica de E-LOTOS (dirigiéndome cuando yo estaba más perdido), y por resolver todas mis dudas sobre LaTeX y Linux. Gracias también a Yolanda Ortega, por haber estado siempre dispuesta a ayudarme, y por sus sugerencias para mejorar la presentación de esta tesis. La ayuda que ella y Narciso me han prestado en mi trabajo como docente ha sido tremenda. Ahora tenemos que terminar de escribir juntos un famoso libro de problemas. También quiero agradecer a mis dos compañeras por haberme aguantado: a Ana Gil, por sus buenos consejos, y a Clara Segura, por compartir tantas cosas conmigo. Gracias también a todos "los peques" de mi grupo, y a los no tan peques, por haber pasado juntos tantos buenos momentos, y a Puri Arenas, por contar siempre conmigo.

Del grupo de Maude también hay mucha gente a la que quiero agradecer su ayuda. Especialmente a Francisco Durán, por todas sus respuestas sobre Full Maude, por haberme apoyado siempre y por sus palabras de aliento cuando he estado hecho un lío. Y también por la colaboración que hemos mantenido en relación a Mobile Maude y sus aplicaciones. Gracias también a José Meseguer por haberme expresado siempre, directamente o a través de Narciso, todo su apoyo. Gracias a Isabel Pita, por el trabajo que hemos realizado juntos; a Steven Eker, por haber respondido siempre todas mis pregundas sobre Maude; y a Manuel Clavel, Mark-Oliver Stehr y Peter Ölveczky, por haberme ayudado y respondido a mis preguntas sobre sus trabajos siempre que lo he necesitado. Y muchas gracias a Roberto Bruni por haberme ayudado cuando empezaba a introducirme en el mundo de las estrategias en Maude.

Quiero agradecer también a Carron Shankland todo el trabajo que hemos realizado juntos. Gran parte del trabajo presentado en esta tesis tiene que ver, directa o indirecta-

mente, con ella. Gracias a ella y a Ken Turner por su hospitalidad en mis estancias en la Universidad de Stirling. Me gustaría agradecer también a David Basin, Christine Röeckl y Leonor Prensa su ayuda con el demostrador de teoremas Isabelle.

Agradezco a los miembros del tribunal su rápida disposición a participar en el mismo y el trabajo dedicado a evaluar la tesis.

Por último, muchas gracias a mis mejores amigos, Juan y Susana, por haberme dado tanto la lata para que terminara de escribir la tesis, y por haber escuchado con tanto interés los rollos que les he contado sobre mi trabajo. Y gracias a mi familia, por saber comprenderme y por soportar mis "malos humos", resultantes muchas veces de atender sólo a mi trabajo.

El dibujo mostrado anteriormente se ha tomado prestado de [GHW85].

Muchas gracias a todos.

Índice general

1. Introducción **1**

2. Lógica de reescritura y Maude **11**

2.1. Lógica de reescritura . . . 11

2.1.1. Reflexión en lógica de reescritura . . . 14

2.2. Maude . . . 14

2.2.1. Módulos funcionales . . . 14

2.2.2. Módulos de sistema . . . 17

2.2.3. Jerarquías de módulos . . . 19

2.2.4. Reflexión en Maude . . . 20

2.2.5. Estrategias internas . . . 25

2.2.6. El módulo predefinido `LOOP-MODE` . . . 32

2.3. Full Maude . . . 33

2.3.1. Módulos orientados a objetos . . . 33

2.3.2. Programación parametrizada . . . 37

2.3.3. Extensiones de `META-LEVEL` . . . 38

2.3.4. Restricciones en la sintaxis de Full Maude . . . 39

2.4. Maude 2.0 . . . 40

2.4.1. `META-LEVEL` en Maude 2.0 . . . 43

3. Semánticas operacionales ejecutables **47**

3.1. Semánticas operacionales estructurales . . . 48

3.2. Reglas de inferencia como reescrituras . . . 49

3.3. Transiciones como reescrituras . . . 57

3.4. Maude como marco semántico ejecutable . . . 60

3.4.1. Reglas de inferencia como reescrituras . . . 60

3.4.2. Transiciones como reescrituras . . . 62

3.5. Maude como metalenguaje . . . 63

4. Ejecución y verificación de CCS en Maude **67**
4.1. CCS . . . 68
4.2. Representación de CCS . . . 69
4.3. Semántica de CCS ejecutable . . . 73
4.3.1. Definición de la semántica ejecutable . . . 73
4.3.2. Búsqueda en el árbol de reescrituras . . . 79
4.3.3. Algunos ejemplos . . . 84
4.4. Cómo obtener nuevas clases de resultados . . . 86
4.4.1. Inclusión de metavariables como procesos . . . 86
4.4.2. Más ejemplos . . . 88
4.4.3. Sucesores de un proceso . . . 88
4.5. Extensión de la semántica a trazas . . . 91
4.6. Extensión a la semántica de transiciones débil . . . 92
4.6.1. Definición de la extensión . . . 92
4.6.2. Ejemplo . . . 94
4.7. Lógica modal para procesos CCS . . . 94
4.7.1. Lógica de Hennessy-Milner . . . 94
4.7.2. Implementación . . . 95
4.7.3. Ejemplos . . . 97
4.7.4. Más modalidades . . . 98
4.8. Comparación con Isabelle . . . 100
4.8.1. Comparación con la representación en Maude . . . 104
4.9. Conclusiones . . . 106

5. Implementación de CCS en Maude 2 **107**
5.1. Sintaxis de CCS . . . 108
5.2. Semántica de CCS . . . 109
5.3. Extensión a la semántica de transiciones débiles . . . 113
5.4. Lógica modal de Hennessy-Milner . . . 114
5.5. Conclusiones . . . 117

6. Otros lenguajes de programación **119**
6.1. El lenguaje funcional *Fpl* . . . 119
6.1.1. Semántica de evaluación . . . 121
6.1.2. Semántica de computación . . . 135
6.1.3. Máquina abstracta para *Fpl* . . . 142
6.2. El lenguaje *WhileL* . . . 148
6.2.1. Semántica de evaluación . . . 150

6.2.2. Semántica de computación . . . 154
6.2.3. El lenguaje *GuardL* . . . 157
6.3. El lenguaje Mini-ML . . . 162
6.4. Conclusiones . . . 168

7. Una herramienta para Full LOTOS **171**
7.1. LOTOS . . . 172
7.2. Semántica simbólica para LOTOS . . . 174
7.3. Semántica simbólica de LOTOS en Maude . . . 176
7.3.1. Sintaxis de LOTOS . . . 176
7.3.2. Semántica simbólica de LOTOS . . . 179
7.3.3. Semántica de términos . . . 192
7.3.4. Ejemplo de ejecución . . . 193
7.4. Lógica modal FULL . . . 194
7.5. Traducción de especificaciones ACT ONE . . . 198
7.5.1. Extensión de los módulos . . . 204
7.6. Construcción de la herramienta LOTOS . . . 206
7.6.1. Gramática de la interfaz de la herramienta . . . 206
7.6.2. Procesamiento de la entrada LOTOS . . . 208
7.6.3. Procesamiento de los comandos de la herramienta . . . 210
7.6.4. Tratamiento del estado de la herramienta . . . 212
7.6.5. El entorno de la herramienta LOTOS . . . 216
7.6.6. Ejemplos de ejecución . . . 218
7.7. Comparación con otras herramientas . . . 222
7.7.1. LOTOS con el enfoque de reglas de inferencia como reescrituras . . . 223
7.7.2. Otras herramientas . . . 224
7.8. Conclusiones . . . 225

8. Protocolo de elección de líder de IEEE 1394 **227**
8.1. Descripción informal del protocolo . . . 229
8.2. Descripción del protocolo con comunicación síncrona . . . 230
8.3. Descripción con comunicación asíncrona con tiempo . . . 232
8.3.1. El tiempo en la lógica de reescritura y Maude . . . 233
8.3.2. Segunda descripción del protocolo . . . 235
8.3.3. Tercera descripción del protocolo . . . 240
8.3.4. Ejecución de la especificación con un ejemplo . . . 243
8.4. Análisis exhaustivo de estados . . . 245
8.4.1. Estrategia de búsqueda . . . 245

8.4.2. Utilización de la estrategia con un ejemplo 248
8.5. Demostración formal por inducción 250
8.6. Evaluación 251
8.7. Ejecución de las especificaciones en Maude 2.0 252
8.8. Comparación con descripciones en E-LOTOS 253
8.9. Conclusiones 255

9. Representación de RDF en Maude **257**
9.1. RDF y RDFS 259
9.2. Traducción de RDF/RDFS a Maude 263
9.3. Traducción automática definida en Maude 267
9.4. Mobile Maude 269
9.5. Caso de estudio: compra de impresoras 274
9.6. Conclusiones 280

10. Conclusiones y trabajo futuro **283**
10.1. Semánticas operacionales ejecutables 283
10.2. Protocolos de comunicación 285
10.3. Lenguajes de especificación de la web semántica 286

Bibliografía **289**

Capítulo 1

Introducción

Uno de los objetivos principales de la ingeniería del software es permitir a los desarrolladores la construcción de sistemas que operen de forma fiable a pesar de su complejidad. Una forma de conseguir este objetivo es mediante el uso de *métodos formales*, es decir, lenguajes, técnicas y herramientas con una fuerte base matemática, útiles para especificar dichos sistemas y sus propiedades, y verificarlos de una forma sistemática. Aunque la utilización de métodos formales no garantiza *a priori* la corrección, puede ayudar a incrementar en gran medida nuestro conocimiento de un sistema, revelando inconsistencias, ambigüedades y defectos, que podrían no detectarse de otra manera. De hecho, el principal beneficio del proceso de especificación (escribir las cosas de forma precisa) es intangible: obtener un conocimiento más profundo del sistema especificado. Sin embargo, un producto tangible de este proceso es un artefacto que puede ser analizado formalmente, por ejemplo comprobando su consistencia, o utilizado para derivar otras propiedades del sistema especificado.

Algunos métodos formales están basados en lógicas, que disponen de un conjunto de reglas de inferencia con las cuales se pueden derivar conjuntos de sentencias del lenguaje de especificación a partir de otros conjuntos de sentencias. De esta forma, a partir de la especificación, vista como un conjunto de hechos o afirmaciones, se pueden derivar nuevas propiedades del objeto especificado. Así, las reglas de inferencia proporcionan al usuario del método formal lógico una forma de predecir el comportamiento de un sistema sin tener que construirlo.

Además, algunos métodos formales facilitan especificaciones *ejecutables* en una computadora que pueden jugar un papel muy importante en el proceso de desarrollo de un sistema. Los especificadores pueden obtener una realimentación inmediata acerca de la especificación en sí misma, utilizando la propia especificación como prototipo del sistema, y probando el sistema especificado por medio de la ejecución simbólica de la especificación.

La *lógica de reescritura* [Mes90, Mes92], propuesta por José Meseguer en 1990 como marco de unificación de modelos de computación concurrente, es una lógica para especificar sistemas concurrentes con estado que evolucionan por medio de transiciones.

Una teoría en la lógica de reescritura consiste en una signatura (que a su vez es una teoría ecuacional) y un conjunto de reglas (posiblemente condicionales) de reescritura.

La signatura de la teoría describe una estructura particular para los estados de un sistema (por ejemplo, multiconjunto, árbol binario, etc.) de forma que dichos estados pueden ser distribuidos de acuerdo a la misma. Las reglas de reescritura en la teoría describen qué transiciones locales elementales son posibles en cada estado distribuido, siendo realizadas mediante transformaciones que tienen un efecto local. De esta forma, cada paso de reescritura está formado por una o varias transiciones locales concurrentes.

La lógica de reescritura tiene una semántica operacional y otra denotacional precisas y bien definidas (como veremos en el Capítulo 2), lo que permite que las especificaciones en lógica de reescritura puedan analizarse formalmente de diversas maneras.

Pero además, la lógica de reescritura es ejecutable, por lo que también puede usarse directamente como un lenguaje de amplio espectro para soportar la especificación, el prototipado rápido y la programación declarativa de alto nivel de sistemas concurrentes [LMOM94]. Esta idea ha sido plasmada en el lenguaje multiparadigma Maude cuyos módulos son teorías en la lógica de reescritura [CDE+99, CDE+00a]. Además, el intérprete de Maude utiliza técnicas avanzadas de semicompilación que hacen posible una implementación muy eficiente.

Entre las ventajas de la lógica de reescritura podemos destacar las siguientes:

- *Tiene un formalismo simple*, con solo unas pocas reglas de deducción (véase Sección 2.1) que son fáciles de entender y justificar;
- *Es muy flexible y expresiva*, capaz de representar cambios en sistemas con estructuras muy diferentes;
- *Permite sintaxis definida por el usuario*, con una completa libertad a la hora de elegir los operadores y las propiedades estructurales apropiadas para cada problema;
- *Es intrínsecamente concurrente*, representando transiciones concurrentes y dando soporte para razonar sobre estas;
- *Soporta el modelado de sistemas concurrentes orientados a objetos*, de una forma simple y directa;
- *Es implementable en un lenguaje de amplio espectro*, Maude, que da soporte a especificaciones ejecutables y programación declarativa.

Con respecto a los usos computacionales de la lógica de reescritura, tenemos que esta proporciona un *marco semántico* en el cual se pueden expresar diferentes lenguajes y modelos de computación. En la serie de artículos [Mes92, MOM93, Mes96b] se ha demostrado que una gran variedad de modelos de computación, incluyendo modelos de la concurrencia, pueden expresarse directa y naturalmente como teorías en la lógica de reescritura. Entre tales modelos se encuentran las redes de Petri, los objetos concurrentes y actores, el cálculo CCS de Milner, los objetos con acceso concurrente, la reescritura de grafos y los sistemas de tiempo real.

Las *semánticas operacionales estructurales* [Plo81] son un método formal para presentar de una manera unificada diferentes aspectos de los lenguajes de programación:

semánticas estáticas (para hablar sobre propiedades de las expresiones del lenguaje, como su tipo), semánticas dinámicas (para referirse a la ejecución o evaluación de expresiones del lenguaje), y traducciones (de un lenguaje a otro). La idea general de este tipo de definiciones semánticas es proporcionar axiomas y reglas de inferencia que caractericen los diferentes predicados semánticos definibles sobre una expresión del lenguaje. Por ejemplo, mediante una semántica estática uno puede afirmar que la expresión e tiene tipo τ en el entorno ρ, axiomatizando el predicado mediante juicios

$$\rho \vdash e : \tau$$

donde ρ es una colección de hipótesis sobre los tipos de las variables en e.

En lo que se refiere a las semánticas dinámicas nos encontramos con una gran variedad de estilos, dependiendo de las propiedades del lenguaje que se describe. Para un lenguaje sencillo, es suficiente expresar que la evaluación de una expresión e en un estado s_1 nos da un nuevo estado s_2. El correspondiente predicado puede escribirse como

$$s_1 \vdash e \Rightarrow s_2$$

donde s_1 almacena los valores de los identificadores que aparecen en e.

Una definición semántica es una lista de *axiomas* y *reglas de inferencia* que definen uno de los predicados anteriores. Las reglas son de la forma

$$\frac{H_1 \vdash T_1 \Rightarrow R_1 \quad \cdots \quad H_n \vdash T_n \Rightarrow R_n}{H \vdash T \Rightarrow R} \text{ si } \textit{condición}$$

donde las H_i son *hipótesis* (típicamente entornos que contienen ligaduras entre identificadores y objetos semánticos), los T_i son *términos* (construidos con la sintaxis abstracta del lenguaje), y los R_i son *resultados* (típicamente tipos, valores, o entornos enriquecidos). A una instancia $H_j \vdash T_j \Rightarrow R_j$ se le denomina *secuente*. Los secuentes sobre la línea horizontal son *las premisas* y el secuente bajo la línea es *la conclusión*. La regla puede interpretarse de la siguiente manera: para probar la conclusión $H \vdash T \Rightarrow R$ (aplicando esta regla), antes se tienen que probar todas las premisas. La condición lateral es una expresión booleana y, si estuviera presente, también tiene que ser satisfecha.

En otras palabras, una definición semántica se identifica con una lógica, y razonar con el lenguaje consiste en demostrar teoremas dentro de la lógica.

En [MOM93] se estudia la representación de diferentes semánticas operacionales estructurales por medio de la lógica de reescritura, para lo cual se pueden seguir dos enfoques diferentes. Uno de ellos consiste en representar los juicios de la semántica operacional como términos de una teoría, y representar cada regla semántica como una regla de reescritura que reescribe el conjunto de términos representando las premisas al término que representa la conclusión, o como una regla de reescritura que reescribe la conclusión al conjunto de premisas. A este método le denominaremos *reglas de inferencia como reescrituras*. El otro enfoque consiste en transformar la relación de transición entre estados, que típicamente aparece en este tipo de semánticas operacionales, en una relación de reescritura entre términos que representen dichos estados. En consecuencia, una regla semántica se

representa como una regla de reescritura condicional donde la reescritura principal corresponde a la transición en la conclusión, y las reescrituras en las condiciones corresponden a las transiciones en las premisas. A este método le denominaremos *transiciones como reescrituras*.

Aunque correctas desde un punto de vista teórico, las representaciones presentadas en [MOM93] no son directamente ejecutables en Maude (del cual en aquel momento no existían implementaciones). Tras la aparición de implementaciones de Maude nos propusimos el objetivo principal de esta tesis: extender la idea de la lógica de reescritura y Maude como marco semántico a la idea de *marco semántico ejecutable*.

Este objetivo es alcanzable gracias, en gran parte, a una de las características más importantes de la lógica de reescritura: *la reflexión*. Las reglas de reescritura no admiten una interpretación ecuacional, pues pueden no terminar ni ser confluentes. Por esta razón, hay muchos cómputos diferentes a partir de un estado dado y es crucial utilizar estrategias apropiadas para controlar la ejecución de un módulo. En Maude, tales estrategias no están fuera de la lógica, sino que son *estrategias internas* definidas por reglas de reescritura al metanivel. Esto es posible porque la lógica de reescritura es *reflexiva* [Cla98], pues tiene una *teoría universal* $\mathcal{U}$ que es capaz de representar cualquier teoría finita $\mathcal{R}$ de la lógica de reescritura (incluyendo a la misma $\mathcal{U}$) y a términos t, t' en $\mathcal{R}$ como términos $\overline{\mathcal{R}}$ y $\overline{t}, \overline{t'}$ en $\mathcal{U}$, de forma que

$$\mathcal{R} \vdash t \longrightarrow t' \quad \Leftrightarrow \quad \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Maude soporta de manera eficiente la reflexión a través de un módulo `META-LEVEL`, que permite no solo la definición y ejecución de estrategias internas mediante reglas de reescritura, sino también muchas otras aplicaciones, incluyendo la metaprogramación, al considerar programas o especificaciones como datos, y un álgebra de módulos extensible [Dur99]. De esta manera, las buenas propiedades reflexivas de Maude permiten utilizarlo como *metalenguaje* [CDE+98b] en el que se puede definir la semántica de otros lenguajes, indicar cómo estos tienen que ser ejecutados y contruir herramientas completas que permitan la introducción y ejecución de programas escritos en estos lenguajes.

El objetivo de presentar a Maude como marco semántico ejecutable se va a abordar en esta tesis desde diferentes enfoques. En primer lugar, vamos a obtener representaciones ejecutables de semánticas operacionales estructurales de muy diferentes tipos, tanto en lo que se refiere a los lenguajes especificados (desde álgebras de procesos como CCS y LOTOS, hasta lenguajes funcionales e imperativos), como al tipo concreto de semántica operacional estructural (semánticas de paso largo, paso corto, etc.). Vamos a presentar una metodología de construcción de estas representaciones semánticas ejecutables, y no solo una serie de soluciones *ad hoc* para cada lenguaje representado. Entendemos que las soluciones presentadas para los problemas de ejecutabilidad encontrados en las representaciones originales son suficientemente generales, como prueba el hecho de que hayan podido ser utilizadas constantemente durante toda la tesis. No obstante, no estamos en disposición de afirmar que la metodología sea completamente general, de modo que se pueda aplicar a una semántica operacional cualquiera.

La utilización de la lógica de reescritura y de Maude como marco semántico no se reduce a la representación de semánticas operacionales estructurales. Ya en el artículo [MOM93]

se utiliza Maude para dar semántica a la programación orientada a objetos concurrente, a lenguajes de resolución de restricciones, y a la representación de las ideas de *acción y cambio* resolviendo el *problema del marco* [MOM99]. Más recientemente, se han estudiado las aplicaciones de la lógica de reescritura y de Maude para la especificación y el análisis de protocolos de comunicación [DMT98]. En [DMT00a] se ha propuesto una metodología formal que resulta ser aplicable en contextos más generales, y no solamente en el caso de los protocolos. Esta metodología se estructura como una jerarquía de métodos incrementalmente más potentes, con la idea de no malgastar recursos, dado el coste que puede suponer la verificación matemática completa de un sistema complejo. De esta forma, tiene sentido utilizar métodos costosos y complejos, solamente después de la aplicación de métodos más simples que hayan proporcionado una mejor comprensión, así como posibles mejoras y correcciones a la especificación original. La jerarquía incluye los siguientes métodos:

1. *Especificación formal*, que facilita un modelo formal del sistema, en el que se hayan aclarado las posibles ambigüedades y hecho explícitas posibles suposiciones ocultas.

2. *Ejecución de la especificación*, con el propósito de poder simular y depurar la especificación al realizar varios cómputos.

3. *Análisis formal mediante "model-checking" (análisis exhaustivo de estados)*, con el fin de encontrar errores a base de considerar todos los posibles cómputos de un sistema altamente distribuido y no determinista a partir de un estado inicial dado, hasta alcanzar cierta cota de profundidad. Si el conjunto de estados es finito, este método ya proporciona una demostración formal de corrección con respecto a las propiedades consideradas.

4. *Análisis mediante "narrowing" (estrechamiento)*, en el cual se analizan todos los cómputos de un conjunto posiblemente infinito de estados gracias a la descripción de los mismos mediante técnicas simbólicas.

5. *Demostración formal*, donde se verifica la corrección de propiedades críticas por medio de alguna técnica formal, bien a mano (con lápiz y papel), o bien con la ayuda de herramientas de demostración.

Nuestro segundo enfoque a la presentación de Maude como marco semántico ejecutable contribuye al desarrollo de esta metodología especificando y analizando tres descripciones ejecutables del protocolo de elección de líder dentro de la especificación del bus multimedia en serie IEEE 1394 (conocido como "FireWire"). El protocolo consiste en, dada una red de nodos conexa y acíclica, construir un árbol de expansión, cuya raíz actuará como líder en las siguientes fases de funcionamiento del bus. En nuestras especificaciones haremos especial énfasis en los aspectos relacionados con el tiempo, esenciales para el protocolo.

En la línea de investigación ya mencionada de utilizar la lógica de reescritura y Maude para representar y dar semántica a otros lenguajes, se han realizado trabajos relacionados con lenguajes previos a la definición de la lógica de reescritura, como Mini-ML, Prolog, BABEL o UNITY, y a lenguajes más novedosos como PLAN, UML o DaAgent (más adelante mostraremos referencias a estos y otros trabajos). Como contribución a esta línea de

trabajo, vamos a abordar la dotación de semántica formal a lenguajes de la *web semántica*, mediante la traducción del lenguaje de descripción de recursos web RDF (*Resource Description Framework*) a Maude y su integración con Mobile Maude [DELM00], una extensión de Maude que permite la definición de cómputos móviles, como veremos en el Capítulo 9.

Estructura de la tesis

En el Capítulo 2 presentamos una introducción a la lógica de reescritura y a Maude, mostrando sus principales características y los conceptos utilizados a lo largo de la tesis. El material contenido en este capítulo no es original, sino que ha sido preparado a partir de varios de los trabajos realizados por el grupo de Maude para la difusión del lenguaje.

En el Capítulo 3 se muestran las ideas presentadas por Narciso Martí y José Meseguer [MOM93] sobre la representación de semánticas operacionales estructurales en la lógica de reescritura, y los problemas generales que estas representaciones presentan a la hora de implementarse en las versiones actuales de Maude, dando ideas de las soluciones que se proponen, las cuales se detallarán en los siguientes capítulos. También se dan indicaciones sobre cómo puede utilizarse Maude como *metalenguaje* en el que implementar herramientas completas para la introducción, la ejecución y el análisis de programas en otros lenguajes.

En el Capítulo 4 se muestra el primer caso concreto de aplicación de estas ideas siguiendo el enfoque de las *reglas de inferencia como reescrituras*. En concreto veremos una representación ejecutable tanto de la semántica del álgebra de procesos CCS de Milner [Mil89] como de la lógica modal de Hennessy-Milner para describir capacidades locales de procesos CCS [HM85]. Se muestran los problemas de ejecutabilidad de este tipo de representaciones (básicamente, la existencia de variables nuevas en la parte derecha de una regla de reescritura y la aplicación no determinista de las reglas semánticas) y cómo se pueden resolver explotando las propiedades reflexivas de la lógica de reescritura.

El ejemplo con CCS y la lógica modal de Hennessy-Milner se completa en el Capítulo 5 donde se presentan implementaciones de las dos semánticas siguiendo el enfoque alternativo de las *transiciones como reescrituras* que presenta diversas ventajas frente al enfoque anterior y que permite la nueva versión de Maude 2.0 actualmente en desarrollo.

En el Capítulo 6 se presentan representaciones ejecutables de semánticas operacionales de lenguajes de programación, tanto funcionales como imperativos (incluyendo no determinismo). Dentro de estas semánticas se verán tanto semánticas de evaluación (o de paso largo) como semánticas de computación (o de paso corto). Los ejemplos presentados son los utilizados por Hennessy en su libro sobre semánticas operacionales [Hen90] y por Kahn en su trabajo sobre semánticas naturales [Kah87].

En el Capítulo 7 se presenta una herramienta formal completa basada en una semántica simbólica [CS01] para el álgebra de procesos con tipos de datos Full LOTOS [ISO89], en la que se pueden ejecutar especificaciones escritas utilizando esta técnica formal. Este lenguaje incluye tanto la especificación del comportamiento de sistemas distribuidos concurrentes, como la especificación algebraica de los tipos de datos que estos sistemas utilizan. Por tanto, para implementar nuestra herramienta es necesario no solo represen-

tar las reglas semánticas de Full LOTOS utilizando las ideas presentadas en capítulos anteriores, sino que hace falta tratar especificaciones de tipos abstractos de datos escritas utilizando el lenguaje ACT ONE [EM85], que serán traducidas a módulos funcionales en Maude con la misma semántica, e integrar estas dos partes en un interfaz que permita la fácil utilización de la herramienta para ejecutar las especificaciones introducidas. Con este capítulo concluye la parte de la tesis dedicada al estudio de la implementación en Maude de semánticas operacionales estructurales.

En el Capítulo 8 se muestran tres descripciones a diferentes niveles de abstracción del protocolo de elección de líder dentro de la especificación del bus multimedia en serie del estándar IEEE 1394 [IEE95]. Las descripciones utilizan módulos orientados a objetos de Maude, y dos de ellas incluyen nociones de tiempo, imprescindibles para la correcta especificación del protocolo con cierto nivel de detalle. La corrección del protocolo se demuestra utilizando técnicas de exploración de estados, similares a las utilizadas en el Capítulo 4 para conseguir una representación ejecutable de la semántica de CCS.

El Capítulo 9 se dedica a mostrar cómo se puede dar semántica al lenguaje RDF de especificación de recursos web [LS99], mediante la traducción de documentos RDF a módulos orientados a objetos en Maude. Esta traducción a Maude no solo proporciona una semántica formal a los documentos RDF, sino que permite que los programas que manipulan estos documentos puedan ser expresados en el mismo formalismo. Gracias a las facilidades de metaprogramación de Maude, la traducción puede implementarse en el mismo Maude. Como ejemplo de utilización de la traducción, y de la integración con los programas que utilizan esta información traducida, se muestra una aplicación basada en agentes móviles, por lo que para realizarla se ha utilizado Mobile Maude [DELM00].

Finalmente, en el Capítulo 10 se presentan algunas conclusiones y líneas de trabajo futuro.

En los siguientes capítulos se muestra prácticamente todo el código Maude, al ser este claro y poco extenso, sin embargo habrá partes que se omitan porque no añadan nada nuevo. El código Maude completo, así como el código Isabelle de la Sección 4.8, puede encontrarse en la página web `http://dalila.sip.ucm.es/~alberto/tesis`, listo para ser ejecutado.

Trabajo relacionado

Encontramos en la literatura diversos trabajos dedicados a la representación e implementación de semánticas operacionales. Citaremos aquí varios de los que entendemos que están más relacionados con nuestro trabajo.

Probablemente el trabajo más relacionado con el nuestro es el realizado por Christiano Braga en su tesis doctoral [Bra01], donde se describe un intérprete para especificaciones MSOS [BHMM00] en el contexto de las semánticas operacionales estructurales modulares de Peter Mosses [Mos99]. Para la implementación del intérprete se utiliza la idea de transiciones como reescrituras mencionada anteriormente. Para ello se realiza una extensión de Maude, implementada utilizando las propiedades reflexivas del propio Maude, que permite reglas de reescritura condicionales con reescrituras en las condiciones. Nosotros

utilizaremos la nueva versión de Maude 2.0 cuando usemos el método de transiciones como reescrituras, obteniendo una considerable mejora en la eficiencia.

Utilizando la lógica de reescritura también se han dado definiciones de lenguajes como el lambda cálculo y Mini-ML [MOM93, Ste00], Prolog y lenguajes basados en estrechamiento (*narrowing*) como BABEL [Vit94], el lenguaje UNITY [Mes92], el π-cálculo [Vir96, Ste00, TSMO02], el lenguaje de programación lógico-concurrente GAEA [IMW+97], el lenguaje de programación para redes activas PLAN [WMG00, ST02], el metamodelo de UML [TF00, FT00, FT01], el lenguaje de especificación de protocolos criptográficos CAPSL [DM00a], el sistema de agentes móviles DaAgent [BCM00], y la extensión de Maude para cómputos móviles Mobile Maude [DELM00] (véase Sección 9.4). Para una bibliografía más exhaustiva sobre este tema referimos al lector al artículo [MOM02].

Quizás el primer intento por conseguir implementaciones directas de semánticas operacionales fue Typol [Des88], un lenguaje formal para representar reglas de inferencia y semánticas operacionales. Los programas en Typol se compilan a Prolog para crear comprobadores de tipos ejecutables e intérpretes a partir de sus especificaciones [Des84]. Aunque algunas de nuestras implementaciones siguen en gran medida el estilo de la programación lógica, una gran ventaja al utilizar Maude se obtiene al poder trabajar por un lado con tipos de datos definidos por el usuario y por otro con especificaciones algebraicas módulo axiomas ecuacionales. Además, podríamos utilizar otras estrategias diferentes de la búsqueda en profundidad, aun manteniendo la misma especificación subyacente.

Algunas de las desventajas de Typol son su ineficiencia y el hecho de que la codificación de especificaciones de semánticas operacionales estructurales en Prolog es poco atractiva, debido a la falta de un sistema de tipos apropiado en Prolog (algunos autores han utilizado el lenguaje de orden superior λProlog [FGH+88] para evitar este problema). Por esas razones se diseñó el lenguaje RML (Relational Meta-Language) [Pet94, Pet96], un lenguaje de especificación ejecutable para semánticas naturales. En ese estudio se identificaron propiedades de las especificaciones de semánticas naturales determinables de forma estática que permiten realizar algunas optimizaciones en la implementación. RML tiene un sistema de tipos fuerte al estilo de Standard ML, y soporta reglas de inferencia como las dadas en las semánticas naturales y definiciones de tipos de datos mediante inducción estructural. Las especificaciones en RML se transforman a una representación intermedia, fácilmente optimizable e implementable, siguiendo el estilo de CPS (*Continuation-Passing Style*). Esta representación intermedia termina compilándose a código C eficiente.

También se han utilizado demostradores de teoremas como Isabelle/HOL [NPW02] o Coq [HKPM02] para construir modelos de lenguajes a partir de su semántica operacional. En la Sección 4.8 utilizaremos Isabelle para implementar CCS y la lógica modal de Hennessy-Milner, compararemos este enfoque con el nuestro, y citaremos trabajos relacionados sobre la representación de sistemas de inferencia en el marco lógico Isabelle/HOL, relacionados con CCS. Isabelle/HOL también ha sido utilizado por Nipkow [Nip98] para formalizar semánticas operacionales y denotacionales de lenguajes de programación. Otros marcos lógicos y demostradores de teoremas se han utilizado así mismo para representar sistemas de inferencia. El entorno interactivo de desarrollo de demostraciones Coq, basado en el cálculo de construcciones (*Calculus of Constructions*) extendido con tipos inductivos [Coq88, PM94], ha sido utilizado para representar el π-cálculo [FHS01, Hir97] y el

μ-cálculo [Spr98] aplicado a CCS. Coq se utiliza para codificar semánticas naturales en [Ter95]. En estos trabajos el enfoque es diferente al nuestro, ya que más que conseguir representaciones ejecutables, el énfasis se pone en obtener modelos sobre los que se puedan verificar metapropiedades.

LEGO, un sistema interactivo de desarrollo de demostraciones [LP92], se utiliza en [YL97] para formalizar un sistema de verificación para CCS. El sistema combina la demostración de teoremas con el análisis exhaustivo de estados (*model checking*), utilizando un demostrador de teoremas para reducir o dividir los problemas hasta subproblemas que pueden ser comprobados por un analizador de estados.

Capítulo 2

Lógica de reescritura y Maude

En este capítulo presentamos las principales características de la lógica de reescritura y del lenguaje y sistema Maude basado en ella, mostrando todos los elementos que se utilizan en el resto de la tesis. De Maude describimos tanto la versión 1.0.5 como la versión 2.0, actualmente en desarrollo. También presentamos la extensión de Maude conocida como Full Maude. El contenido de este capítulo se ha adaptado a partir de varios trabajos [CDE+99, MOM93, Dur99, MOM02, CDE+00b].

2.1. Lógica de reescritura

La lógica ecuacional, en todas sus variantes, es una lógica para razonar sobre tipos de datos *estáticos*, donde las nociones de "antes" y "después" no tienen sentido, ya que gracias a la simetría de la igualdad de la lógica ecuacional, todo cambio es reversible. Eliminando la regla de deducción de la simetría de la lógica ecuacional, las ecuaciones dejan de ser simétricas y se convierten en orientadas, como en la reescritura ecuacional. La *lógica de reescritura* [Mes92, Mes98] va un paso (muy importante) más allá eliminando la simetría y la interpretación ecuacional de las reglas, e interpretando una regla $t \longrightarrow t'$

- *computacionalmente*, como una *transición local* en un sistema concurrente; es decir, t y t' describen patrones de *fragmentos* del estado distribuido de un sistema, y la regla explica cómo puede tener lugar una transición local concurrente en tal sistema, cambiando el fragmento local del estado de una instancia del patrón t a la correspondiente instancia del patrón t';
- *lógicamente*, como una *regla de inferencia*, de tal forma que se pueden inferir fórmulas de la forma t' a partir de fórmulas de la forma t.

La lógica de reescritura es una lógica *de cambio*, que nos permite especificar los aspectos dinámicos de los sistemas en un sentido muy general. Es más, permitiendo la reescritura sobre clases de equivalencia módulo algunos axiomas estructurales, podemos entender un "término" $[t]$ como una *proposición* o *fórmula* que afirma estar en un cierto *estado* con una cierta *estructura*.

Una *signatura* en lógica de reescritura es una teoría ecuacional[1] (Σ, E), donde Σ es una signatura ecuacional, esto es, un alfabeto de símbolos de función con su rango correspondiente, y E es un conjunto de Σ-ecuaciones. En la signatura (Σ, E) se hace explícito el conjunto de ecuaciones para hacer énfasis en el hecho de que la reescritura tendrá lugar sobre clases de términos congruentes *módulo* E.

Dada una signatura (Σ, E), las *sentencias* de la lógica son secuentes (llamados *reescrituras*) de la forma

$$[t]_E \longrightarrow [t']_E,$$

donde t y t' son Σ-términos posiblemente con variables, y $[t]_E$ denota la clase de equivalencia del término t módulo las ecuaciones en E. El subíndice E se omite cuando no hay lugar a ambigüedad.

Una *teoría de reescritura* $\mathcal{R}$ es una tupla $\mathcal{R} = (\Sigma, E, L, R)$, donde (Σ, E) es una signatura, L es un conjunto de etiquetas, y R es un conjunto de *reglas de reescritura* de la forma

$$r : [t] \longrightarrow [t'] \ \textit{if} \ [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k],$$

donde r es una etiqueta y se reescriben clases de congruencia de términos en $\mathcal{T}_{\Sigma,E}(X)$, con $X = \{x_1, \ldots, x_n, \ldots\}$ un conjunto infinito contable de variables.

Dada una teoría de reescritura $\mathcal{R}$, decimos que de $\mathcal{R}$ se *deduce* una sentencia $[t] \longrightarrow [t']$, o que $[t] \longrightarrow [t']$ es una $\mathcal{R}$-reescritura (concurrente), y lo escribimos como $\mathcal{R} \vdash [t] \longrightarrow [t']$, si y solo si $[t] \longrightarrow [t']$ puede obtenerse con un número finito de aplicaciones de las siguientes *reglas de deducción* (donde suponemos que todos los términos están bien formados y $t(\overline{w}/\overline{x})$ denota la sustitución simultánea de cada x_i por el correspondiente w_i en t):

1. **Reflexividad**. Para cada $[t] \in \mathcal{T}_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}.$$

2. **Congruencia**. Para cada $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \ \ldots \ [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

3. **Reemplazamiento**. Para cada regla $r : [t] \longrightarrow [t']$ *if* $[u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k]$ en R

$$\frac{\begin{array}{c}[w_1] \longrightarrow [w'_1] \ \ldots \ [w_n] \longrightarrow [w'_n] \\ [u_1(\overline{w}/\overline{x})] \longrightarrow [v_1(\overline{w}/\overline{x})] \ \ldots \ [u_k(\overline{w}/\overline{x})] \longrightarrow [v_k(\overline{w}/\overline{x})]\end{array}}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

4. **Transitividad**.

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

[1] La lógica de reescritura está parametrizada sobre la lógica ecuacional subyacente, pudiendo ser esta la lógica ecuacional sin tipos, con tipos, con tipos ordenados, de pertenencia, etc.

Una sentencia $[t] \longrightarrow [t']$ en $\mathcal{R}$ se denomina una *reescritura en un paso* si y solo si puede ser derivada a partir de $\mathcal{R}$ con un número finito de aplicaciones de las reglas (1)–(3), con al menos una aplicación de la regla (3). Si la regla (3) se aplica exactamente una vez, entonces se dice que la sentencia es una *reescritura secuencial en un paso*.

La lógica de reescritura es una lógica para razonar de forma correcta sobre *sistemas concurrentes* que tienen *estados*, y evolucionan por medio de *transiciones*. La signatura de una teoría de reescritura describe una estructura particular para los estados de un sistema (por ejemplo, multiconjuntos, secuencias, etc.) de modo que los estados pueden distribuirse de acuerdo con tal estructura. Las reglas de reescritura en la teoría describen por medio de transformaciones locales concurrentes qué *transiciones locales elementales* son posibles en el estado distribuido. Las reglas de deducción de la lógica de reescritura nos permiten razonar sobre qué transiciones concurrentes *generales* son posibles en un sistema que satisfaga tal descripción. De forma alternativa, podemos adoptar un punto de vista lógico, y ver las reglas de deducción de la lógica de reescritura como *metarreglas* para la deducción correcta en un *sistema lógico*.

Los puntos de vista computacional y lógico bajo los cuales se puede interpretar la lógica de reescritura pueden resumirse en el siguiente diagrama de correspondencias:

Estado	$\leftrightarrow$	Término	$\leftrightarrow$	Proposición
Transición	$\leftrightarrow$	Reescritura	$\leftrightarrow$	Deducción
Estructura distribuida	$\leftrightarrow$	Estructura algebraica	$\leftrightarrow$	Estructura proposicional

Además de tener un sistema de inferencia, la lógica de reescritura también tiene una *teoría de modelos* con interpretaciones naturales, tanto computacional como lógicamente. Es más, cada teoría de reescritura $\mathcal{R}$ tiene un *modelo inicial* $\mathcal{T}_{\mathcal{R}}$ [Mes92]. La idea es que se pueden decorar los secuentes probables con *términos de demostración* que indiquen cómo pueden ser probados. Desde un punto de vista computacional, un término de demostración es una descripción de un cómputo concurrente, posiblemente complejo; desde un punto de vista lógico, es una descripción de una deducción lógica. La cuestión es, ¿cuándo dos términos de demostración deberían ser considerados descripciones *equivalentes* del mismo cómputo/deducción? Con el modelo $\mathcal{T}_{\mathcal{R}}$ se responde a esta pregunta igualando términos de demostración de acuerdo a unas ecuaciones de equivalencia naturales [Mes92]. De esta forma, se obtiene un modelo $\mathcal{T}_{\mathcal{R}}$ con una estructura de categoría, donde los objetos son clases de equivalencia (módulo las ecuaciones E de $\mathcal{R}$) de Σ-términos cerrados, y las flechas son clases de equivalencia de términos de demostración. Las identidades se asocian naturalmente con demostraciones por reflexividad, y la composición de flechas corresponde a demostraciones por transitividad. Las interpretaciones lógica y computacional son entonces obvias, ya que una categoría es un sistema de transiciones estructurado, y los sistemas lógicos se han entendido como categorías desde el trabajo de Lambek [Lam69] sobre sistemas deductivos. La teoría de demostración y la teoría de modelos de la lógica de reescritura están relacionadas por un *teorema de completitud*, que establece que un secuente se puede probar a partir de $\mathcal{R}$ si y solo si es satisfecho en todos los modelos de $\mathcal{R}$ [Mes92].

2.1.1. Reflexión en lógica de reescritura

La lógica de reescritura es *reflexiva* en una forma matemáticamente precisa [Cla00], a saber, existe una teoría de reescritura $\mathcal{U}$ finitamente representable que es *universal* en el sentido de que cualquier teoría de reescritura $\mathcal{R}$ finitamente representable (incluyendo la propia $\mathcal{U}$) puede ser representada en $\mathcal{U}$ como un término $\overline{\mathcal{R}}$, cualquier término t en $\mathcal{R}$ puede representarse como un término $\overline{t}$, y cualquier par $(\mathcal{R}, t)$ puede representarse como un término $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, de tal forma que tengamos la siguiente equivalencia

$$\mathcal{R} \vdash t \longrightarrow t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Puesto que $\mathcal{U}$ puede representarse a sí misma, podemos conseguir una "torre reflexiva" con un número arbitrario de niveles de reflexión, ya que tenemos

$$\begin{aligned} & \mathcal{R} \vdash t \longrightarrow t' \\ \Leftrightarrow\; & \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\ \Leftrightarrow\; & \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \\ & \ldots \end{aligned}$$

En esta cadena de equivalencias se dice que el primer cómputo de reescritura tiene lugar en el nivel 0 (o nivel objeto), el segundo en el nivel 1 (o metanivel), y así sucesivamente.

2.2. Maude

Maude es tanto un lenguaje de alto nivel como un sistema eficiente que admite especificaciones tanto de la lógica ecuacional de pertenencia como de la lógica de reescritura, y la programación de un espectro muy amplio de aplicaciones. En las siguientes secciones se describen las características más importantes del lenguaje, en su versión Maude 1.0.5, que serán utilizadas en el resto de esta tesis. En la actualidad se está desarrollando una nueva versión del sistema Maude, conocida como Maude 2.0, más general y expresiva. Describiremos esta nueva versión en la Sección 2.4.

El sistema Maude (en su versión 1.0.5), su documentación, una colección de ejemplos, algunos casos de estudio, y varios artículos relacionados están disponibles en la página web de Maude en `http://maude.cs.uiuc.edu`.

2.2.1. Módulos funcionales

Los *módulos funcionales* en Maude definen tipos de datos y funciones sobre ellos por medio de teorías ecuacionales cuyas ecuaciones son Church-Rosser y terminantes. Un módelo matemático de los datos y las funciones viene dado por el *álgebra inicial* definida por la teoría, cuyos elementos son las clases de equivalencia de términos cerrados módulo las ecuaciones.

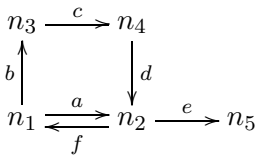


Figura 2.1: Un grafo.

La lógica ecuacional en la cual se basan los módulos funcionales en Maude es una extensión de la lógica ecuacional con tipos ordenados denominada *lógica ecuacional de pertenencia* [BJM00]. Además de tipos, relaciones de subtipado, y sobrecarga en los símbolos de función, los módulos funcionales pueden tener *axiomas (condicionales) de pertenencia*, en los cuales se afirma que un término tiene cierto tipo (si se cumplen ciertas condiciones). Estos axiomas de pertenencia pueden utilizarse para definir funciones parciales, que quedan definidas cuando sus argumentos satisfacen ciertas condiciones ecuacionales o de pertenencia.

Podemos ilustrar estas ideas con un módulo `PATH` que define caminos en un grafo. Considérese el grafo de la Figura 2.1. Un camino en el grafo es una concatenación de aristas tales que el nodo destino de una arista es el nodo origen de la siguiente. Por supuesto, no todas las concatenaciones de aristas son caminos válidos. El siguiente módulo `PATH` axiomatiza el grafo y los caminos sobre él[2].

```
fmod PATH is
  protecting MACHINE-INT .

  sorts Edge Path Path? Node .
  subsorts Edge < Path < Path? .

  ops n1 n2 n3 n4 n5 : -> Node .
  ops a b c d e f : -> Edge .
  op _;_ : Path? Path? -> Path? [assoc] .
  ops source target : Path -> Node .
  op length : Path -> MachineInt .

  var E : Edge .
  var P : Path .

  cmb (E ; P) : Path if target(E) == source(P) .

  ceq source(E ; P) = source(E) if E ; P : Path .
  ceq target(P ; E) = target(E) if P ; E : Path .
  eq length(E) = 1 .
  ceq length(E ; P) = 1 + length(P) if E ; P : Path .

  eq source(a) = n1 .    eq target(a) = n2 .
```

[2]En la Sección 2.4 veremos este mismo módulo con la nueva sintaxis de Maude 2.0 generalizada para permitir mayor soporte de la lógica ecuacional de pertenencia.

```
  eq source(b) = n1 .    eq target(b) = n3 .
  eq source(c) = n3 .    eq target(c) = n4 .
  eq source(d) = n4 .    eq target(d) = n2 .
  eq source(e) = n2 .    eq target(e) = n5 .
  eq source(f) = n2 .    eq target(f) = n1 .
endfm
```

El módulo se introduce con la sintaxis de un módulo funcional `fmod...endfm` y tiene un nombre, `PATH`. Importa el tipo predefinido de enteros `MACHINE-INT` y declara tipos (`sorts`) y relaciones de subtipado (`subsorts`). Una relación de subtipado entre dos tipos se interpreta como una inclusión conjuntista, es decir, los elementos del subtipo se incluyen en el supertipo. Por ejemplo, la declaración de subtipado

```
  subsorts Edge < Path < Path? .
```

declara las aristas como un subtipo de los caminos, y estos como un subtipo del tipo `Path?`, que podríamos llamar de *caminos confusos*. Este supertipo es necesario porque en general la concatenación de caminos utilizando el operador `_;_` puede generar concatenaciones sin sentido. Este operador se declara con sintaxis *infija* (los símbolos de subrayado indican las posiciones de los argumentos) y con el atributo `assoc` que indica que el operador es *asociativo*. Esto significa que el encaje de patrones de términos construidos con este operador se hará módulo asociatividad, es decir, sin que importen los paréntesis (que pueden no escribirse explícitamente). En general, el motor de reescritura de Maude puede reescribir *módulo* la mayoría de las diferentes combinaciones de asociatividad, conmutatividad (`comm`), identidad (por la izquierda, `left id:`; por la derecha, `right id:`; o por ambos lados, `id:`) e idempotencia (`idem`)[3].

La eliminación de concatenaciones sin sentido se lleva a cabo por medio del axioma de pertenencia condicional

```
  cmb (E ; P) : Path if target(E) == source(P) .
```

que establece que una arista concatenada con un camino es también un camino si el nodo destino de la arista coincide con el nodo origen del camino.

Todas las variables en el lado derecho de cada ecuación tienen que aparecer en el correspondiente lado izquierdo, y las variables en las condiciones de cada ecuación (axioma de pertenencia) tienen que aparecer en el correspondiente lado izquierdo (predicado de pertenencia). Esta restricción solo se aplica si queremos ejecutar la especificación.

Las expresiones formadas por los operadores declarados en un módulo pueden evaluarse con el comando de Maude `reduce` (`red` de forma abreviada). En el proceso de reducción las ecuaciones se utilizan de izquierda a derecha como reglas de simplificación, y los axiomas de pertenencia se utilizan para conocer el menor tipo de una expresión.

[3] Los axiomas ecuacionales declarados como atributos de operadores *no* deben declararse como ecuaciones. Ello es así por dos razones: primero porque sería redundante, y segundo porque aunque la semántica denotacional no se vería afectada, sí se modificaría la *semántica operacional*, pues puede afectar a la terminación de la especificación. No obstante, el problema de falta de terminación puede aparecer también en presencia de atributos. En general, el especificador tiene que ser cuidadoso con la interacción entre atributos y ecuaciones.

```
Maude> reduce  b ; c ; d .
result Path: b ; c ; d

Maude> red length(b ; c ; d) .
result NzMachineInt: 3
```

Maude tiene una librería de módulos predefinidos que, por defecto, son cargados en el sistema al principio de cada sesión. Estos módulos son `BOOL`, `MACHINE-INT`, `QID`, `QID-LIST`, `META-LEVEL` y `LOOP-MODE`.

El módulo `BOOL` define los valores booleanos `true` y `false` y algunos operadores como la igualdad `_==_`, la desigualdad `_=/=_`, `if_then_else_fi`, y algunos de los operadores booleanos usuales, como la conjunción (`_and_`), disyunción (`_or_`), negación (`not_`), etc.

El módulo `MACHINE-INT` proporciona un tipo `MachineInt` de enteros predefinidos, un tipo `NzMachineInt` de enteros distintos de 0, y las operaciones aritméticas habituales para trabajar con enteros.

El módulo `QID` proporciona el tipo `Qid` de identificadores con comilla (al estilo de LISP), junto con operaciones sobre estos identificadores, como la concatenación (`conc`), la indexación de un identificador por un entero (`index`), o la eliminación del primer carácter después de la comilla (`strip`). Los siguientes ejemplos muestran el comportamiento de estas operaciones.

```
conc('a, 'b) = 'ab
conc('a, '42) = 'a42
index('a, 2 * 21) = 'a42
conc('a, index(' , 1 - 43)) = 'a-42
strip('abcd) = 'bcd
```

Resulta también útil disponer de un tipo de datos de listas de identificadores con comilla. El módulo `QID-LIST` extiende al módulo `QID` ofreciendo un tipo `QidList` de listas de identificadores con comilla.

```
fmod QID-LIST is
  protecting QID .
  sort QidList .
  subsort Qid < QidList .
  op nil : -> QidList .
  op __ : QidList QidList -> QidList [assoc id: nil] .
endfm
```

El módulo `META-LEVEL` se explicará en la Sección 2.2.4, y el módulo `LOOP-MODE` en la Sección 2.2.6.

2.2.2. Módulos de sistema

Los módulos más generales de Maude son los *módulos de sistema*, que representan una teoría de la lógica de reescritura. Además de los elementos declarables en un módulo funcional, un módulo de sistema puede incluir reglas etiquetadas de reescritura, posiblemente

condicionales. Si consideramos el grafo de la Figura 2.1 como un sistema de transiciones: los nodos se convierten en estados y las aristas en transiciones entre estados. El siguiente módulo de sistema especifica un sistema de transiciones como una teoría de reescritura. Cada transición se convierte en una regla de reescritura, donde el nombre de la transición etiqueta la correspondiente regla.

```
mod A-TRANSITION-SYSTEM is
  sort State .
  ops n1 n2 n3 n4 n5 : -> State .

  rl [a] : n1 => n2 .
  rl [b] : n1 => n3 .
  rl [c] : n3 => n4 .
  rl [d] : n4 => n2 .
  rl [e] : n2 => n5 .
  rl [f] : n2 => n1 .
endm
```

Obsérvese que esta especificación no es confluente, ya que, por ejemplo, hay dos transiciones que salen del nodo `n2` que no pueden unirse, y tampoco es terminante, ya que hay ciclos que crean cómputos infinitos. Por tanto, y en oposición a la simplificación ecuacional en módulos funcionales, la reescritura puede ir en muchas direcciones. Sin embargo, la reescritura puede ser controlada por el usuario por medio de *estrategias*. El intérprete de Maude proporciona una estrategia por defecto para la ejecución de expresiones en módulos de sistema a través del comando `rewrite` (`rew` en forma abreviada). Debido a la posibilidad de no terminación, este comando admite un argumento adicional que acota el número de aplicaciones de reglas. Por ejemplo,

```
Maude> rew [10] n3 .
result State: n5
```

La teoría de reescritura (Σ, E, L, R) correspondiente a un módulo de sistema tiene una signatura Σ dada por los tipos, relaciones de subtipado y operadores declarados, y un conjunto E de ecuaciones, que se asume que puede ser descompuesto en una unión $E = A \cup E'$, donde A es un conjunto de axiomas (entre aquellos soportados por Maude) módulo los cuales se hace la reescritura, y E' es un conjunto de ecuaciones terminantes y Church-Rosser módulo A.

Una regla de reescritura puede contener variables *nuevas* en el lado derecho que no aparezcan en el lado izquierdo. Sin embargo, ya que la aplicación práctica de tales reglas requiere información adicional sobre cómo instanciar estas variables, las reglas con variables nuevas no pueden ser utilizadas por el comando por defecto `rewrite`, sino que deben ser ejecutadas al metanivel. En la versión Maude 1.0.5 las condiciones de una regla condicional tienen que cumplir las mismas restricciones que las condiciones de una ecuación, incluyendo el hecho de que todas sus variables deben aparecer en el lado izquierdo de la regla, si queremos ejecutar la especificación al nivel objeto (véase Sección 2.2.4). Trabajando al metanivel se puede evitar esta restricción, como veremos en el Capítulo 4. Esta restricción se ha relajado considerablemente en Maude 2.0 (véase la Sección 2.4).

Como un ejemplo más de módulo de sistema veamos el siguiente módulo `SORTING` para ordenar vectores de enteros. Los vectores se representan como conjuntos de pares de enteros, donde la primera componente de cada par representa una posición (o índice) del vector y la segunda el valor en dicha posición.

```
mod SORTING is
  protecting MACHINE-INT .

  sorts Pair PairSet .
  subsort Pair < PairSet .

  op <_;_> : MachineInt MachineInt -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .

  vars I J X Y : MachineInt .

  crl [sort] : < J ; X > < I ; Y > => < J ; Y > < I ; X >
            if (J < I) and (X > Y) .
endm
```

Los estados son, por tanto, conjuntos P de pares de enteros, a saber, elementos del tipo `PairSet`. Para facilitar el ejemplo, asumiremos que todo par `<` i `;` x `>` en un conjunto de entrada P verifica que $1 \leq i \leq \text{card}(P)$ y que no puede haber pares diferentes `<` i `;` x `>` y `<` j `;` y `>` con $i = j$. En tal caso, un conjunto de entrada P se puede ver como un vector de enteros[4].

El módulo tiene una única regla condicional `sort`, que modifica un vector de enteros para ordenarlo. El sistema descrito es, por tanto, altamente concurrente, ya que la regla `sort` se puede aplicar de forma concurrente a muchas parejas de pares del conjunto que representa el vector. Utilizando el comando `rew` podemos utilizar el intérprete por defecto de Maude para ordenar un vector de enteros:

```
Maude> rew < 1 ; 3 > < 2 ; 2 > < 3 ; 1 > .
result PairSet: < 1 ; 1 > < 2 ; 2 > < 3 ; 3 >
```

Utilizando este intérprete no tenemos ningún control sobre la aplicación de las reglas en un módulo. Aunque en este caso esto no representa un problema, ya que esta especificación es confluente y terminante, en general puede ocurrir que queramos, y debamos de hecho, controlar la forma en la que se aplican las reglas. Esto puede hacerse mediante la introducción de *estrategias*, como veremos en la Sección 2.2.5.

2.2.3. Jerarquías de módulos

Las especificaciones y el código Maude deberían estructurarse en módulos de tamaño relativamente pequeño para facilitar la comprensión de sistemas de gran tamaño, incre-

[4] Por supuesto, estos requisitos podrían ser especificados de forma explícita declarando un subtipo y axiomas de pertenencia que impongan tales restricciones.

mentar la reutilización de componentes, y localizar los efectos producidos por cambios en el sistema. Full Maude ofrece un soporte completo a estos objetivos, como se verá en la Sección 2.3, facilitando un *álgebra de módulos* extensible, que soporta en particular la programación parametrizada. Sin embargo, (Core) Maude ya proporciona un soporte básico a la modularidad permitiendo la definición de *jerarquías* de módulos, es decir, grafos acíclicos de importaciones de módulos. Matemáticamente, se pueden entender tales jerarquías como órdenes parciales de inclusiones de teorías, es decir, la teoría del módulo importador contiene a las teorías de los submódulos como subteorías.

Maude ofrece dos tipos de inclusiones de módulos, introducidas por las palabras clave `including` y `protecting`, seguidas de una lista no vacía de módulos importados. La importación de tipo `including` es la forma más general, y no impone ningún requisito a la importación. La importación `protecting` es más restrictiva, en el sentido de que hace una *aserción semántica* sobre la relación entre las dos teorías. De forma intuitiva, la aserción afirma que el supermódulo no añade ni "basura" ni "confusión" al submódulo, es decir, no se crean nuevos valores de los tipos definidos en los módulos importados ni se identifican valores de estos tipos que fueran diferentes.

2.2.4. Reflexión en Maude

En Maude, la principal funcionalidad de la reflexión se ha implementado de forma eficiente en el módulo funcional `META-LEVEL`. En este módulo

- los términos Maude se representan como elementos de un tipo de datos `Term` de términos;
- los módulos Maude se representan como términos en un tipo de datos `Module` de módulos;
- el proceso de reducción de un término a su forma normal en un módulo funcional y de determinar si tal forma normal tiene un tipo dado se representan por una función `meta-reduce`;
- el proceso de aplicación de una regla de un módulo de sistema a un término se representa por medio de una función `meta-apply`;
- el proceso de reescritura de un término en un módulo de sistema utilizando el intérprete por defecto de Maude se representa por una función `meta-rewrite`; y
- el análisis sintáctico y la impresión edulcorada (en inglés, *pretty-printing*) de un término en un módulo, así como operaciones sobre tipos como la comparación de tipos en el orden de subtipado de una signatura, son representadas también por funciones adecuadas al metanivel.

Sintaxis de términos y módulos

Los términos se representan como elementos del tipo `Term`, construidos con la siguiente signatura:

```
  subsort Qid < Term .
  subsort Term < TermList .

  op {_}_ : Qid Qid -> Term .
  op _[_] : Qid TermList -> Term .
  op _,_ : TermList TermList -> TermList [assoc] .
  op error* : -> Term .
```

La primera declaración, que convierte a `Qid` en un subtipo de `Term`, se utiliza para representar las variables como identificadores con comilla. El operador `{_}_` se utiliza para representar las constantes como pares, donde el primer argumento es la constante y el segundo su tipo, siendo ambos identificadores con comilla. El operador `_[_]` corresponde a la construcción recursiva de términos a partir de subtérminos, con el operador más externo como primer argumento y la lista de sus subtérminos como segundo argumento, donde la concatenación de listas se denota por `_,_`. La última declaración para el tipo de datos de los términos es una constante `error*` utilizada para representar valores erróneos.

Para ilustrar esta sintaxis, utilizaremos un módulo `NAT` de números naturales con cero y sucesor, y operadores de suma y multiplicación conmutativos.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s N + M = s (N + M) .
  eq 0 * N = 0 .
  eq s N * M = M + (N * M) .
endfm
```

El término `s s 0 + s N` de tipo `Nat` en el módulo `NAT` se metarrepresenta como

```
'_+_['s_['s_[{'0}'Zero]], 's_['N]].
```

Ya que los términos del módulo `META-LEVEL` pueden metarrepresentarse como los términos de cualquier otro módulo, la representación de términos puede iterarse. Por ejemplo, la meta-metarrepresentación $\overline{\overline{\texttt{s 0}}}$ del término `s 0` en `NAT` es el término

```
'_[_][{''s_}'Qid,'{_}_[{''0}'Qid,{''Zero}'Qid]].
```

Los módulos funcionales y de sistema se metarrepresentan con una sintaxis muy similar a la original. Las principales diferencias son que: (1) los términos en ecuaciones, axiomas de pertenencia y reglas de reescritura aparecen metarrepresentados, según la sintaxis ya explicada; (2) en la metarrepresentación de módulos se sigue un orden fijo en la introducción de las diferentes clases de declaraciones; y (3) los conjuntos de identificadores

utilizados en las declaraciones de tipos se representan como conjuntos de identificadores con comilla construidos con el operador asociativo y conmutativo `_;_`.

La sintaxis de los operadores más externos para la representación de módulos funcionales y de sistema es la siguiente:

```
sorts FModule Module .
subsort FModule < Module .

op fmod_is_______endfm : Qid ImportList SortDecl SubsortDeclSet
      OpDeclSet VarDeclSet MembAxSet EquationSet -> FModule .

op mod_is________endm : Qid ImportList SortDecl SubsortDeclSet
      OpDeclSet VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

La definición completa de esta sintaxis en el módulo `META-LEVEL` puede encontrarse en [CDE+99].

La representación $\overline{\texttt{NAT}}$ del módulo `NAT` es el siguiente término de tipo `FModule`:

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
  var 'N : 'Nat .
  var 'M : 'Nat .
  none
  eq '_+_[{'0}'Zero, 'N] = 'N .
  eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
  eq '_*_[{'0}'Zero, 'N] = {'0}'Zero .
  eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N, 'M]] .
endfm
```

Obsérvese que, como cualesquiera términos, los términos de tipo `Module` pueden ser metarrepresentados de nuevo, dando lugar a términos de tipo `Term`, y esta metarrepresentación puede iterarse un número arbitrario de veces. Esto es de hecho necesario cuando los cómputos al metanivel tienen que operar a niveles más altos, como se verá en la Sección 4.7.

Funciones de descenso

El módulo `META-LEVEL` tiene tres funciones predefinidas (implementadas internamente por el sistema) que proporcionan formas eficientes y útiles de reducir cómputos del metanivel a cómputos del nivel objeto (nivel 0): `meta-reduce`, `meta-apply` y `meta-rewrite`. Estas funciones se denominan *funciones de descenso* [CDE+98a].

La operación `meta-reduce`, con sintaxis

```
op meta-reduce : Module Term -> Term .
```

toma como argumentos la representación de un módulo M y un término t, y devuelve la forma completamente reducida del término t utilizando las ecuaciones en M. Por ejemplo[5],

```
Maude> red meta-reduce(NAT, s 0 + s 0) .
result Term: s s 0
```

La operación `meta-rewrite` tiene sintaxis

```
op meta-rewrite : Module Term MachineInt -> Term .
```

y es análoga a `meta-reduce`, pero en vez de utilizar solo la parte ecuacional de un módulo, utiliza tanto las ecuaciones como las reglas para reescribir un término utilizando la estrategia por defecto de Maude. Sus primeros dos argumentos son las representaciones de un módulo M y un término t, y su tercer argumento es un número natural n. La función devuelve la representación del término obtenido a partir de t después de, como mucho, n aplicaciones de reglas de M, utilizando la estrategia por defecto de Maude, que aplica las reglas de forma justa de arriba abajo. Si se da el valor 0 como tercer argumento, el número de reescrituras no estaría acotado.

Utilizando la metarrepresentación del módulo `A-TRANSITION-SYSTEM` visto en la Sección 2.2.2, podemos reescribir (utilizando como mucho 10 aplicaciones de reglas) la metarrepresentación del estado `n3`:

```
Maude> red meta-rewrite(A-TRANSITION-SYSTEM, n3, 10) .
result Term: n5
```

La operación `meta-apply` tiene como perfil

```
op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair .
```

Los primeros cuatro argumentos son las representaciones de un módulo M, un término t en M, una etiqueta l de alguna(s) regla(s) en M y un conjunto de asignaciones (posiblemente vacío) que definen una sustitución parcial σ de las variables en esas reglas. El último argumento es un número natural n. Esta función devuelve una pareja de tipo `ResultPair` formada por un término y una sustitución. La sintaxis de las sustituciones y de los resultados es

```
sorts Assignment Substitution ResultPair .
subsort Assignment < Substitution .
```

[5]Para simplificar la presentación utilizamos la metanotación $\overline{t}$ para denotar la metarrepresentación del término t y $\overline{Id}$ para denotar la metarrepresentación del módulo con nombre Id. Como veremos en la Sección 2.3, en Full Maude podremos utilizar la función `up` para obtener estas metarrepresentaciones.

```
op _<-_ : Qid Term -> Assignment .
op none : -> Substitution .
op _;_ : Substitution Substitution -> Substitution [assoc comm id: none] .

op {_,_} : Term Substitution -> ResultPair .
```

La operación `meta-apply` se evalúa de la siguiente manera:

1. el término t se reduce completamente utilizando las ecuaciones en M;
2. el término resultante se intenta encajar con los lados izquierdos de todas las reglas con etiqueta l parcialmente instanciadas con σ, descartando aquellos encajes que no satisfagan la condición de la regla;
3. se descartan los primeros n encajes que hayan tenido éxito; si existe un $n+1$-ésimo encaje, su regla se aplica utilizando este encaje y se realizan los pasos siguientes 4 y 5; en caso contrario, se devuelve el par `{error*, none}`;
4. el término resultante se reduce completamente utilizando las ecuaciones de M;
5. se devuelve el par formado utilizando el constructor `{_,_}` cuyo primer elemento es la representación del término resultante completamente reducido y cuyo segundo elemento es la representación del encaje utilizado en la reescritura.

Utilizando la operación `meta-apply` y el módulo `A-TRANSITION-SYSTEM` metarrepresentado, se pueden obtener todas las *reescrituras secuenciales en un paso* del estado `n2`:

```
Maude> red meta-apply('A-TRANSITION-SYSTEM, 'n2, 'e, none, 0) .
result ResultPair: { 'n5, none }

Maude> red meta-apply('A-TRANSITION-SYSTEM, 'n2, 'f, none, 0) .
result ResultPair: { 'n1, none }
```

Además de estas tres funciones, el módulo `META-LEVEL` proporciona otras funciones pertenecientes a la teoría universal y que podrían haber sido definidas ecuacionalmente, pero que por razones de eficiencia se han implementado directamente. Estas funciones incluyen el análisis sintáctico, la impresión edulcorada de términos de un módulo al metanivel y operaciones sobre los tipos declarados en la signatura de un módulo.

La operación `meta-parse` tiene como perfil

```
op meta-parse : Module QidList -> Term .
```

Esta operación toma como argumentos la representación de un módulo M y la representación de una lista de *tokens* (componentes sintácticas) dada por una lista de identificadores con comilla, y devuelve el término analizado a partir de la lista de tokens para la signatura de M. Si la lista no se puede analizar, se devuelve la constante `error*`. Por ejemplo, dado el módulo `NAT` y la entrada `'s '0 '+ 's '0`, obtenemos el siguiente resultado:

```
Maude> red meta-parse(NAT, 's '0 '+ 's '0) .
result Term : '_+_['s_[{'0}'Zero], 's_[{'0}'Zero]]
```

La operación `meta-pretty-print` tiene como perfil

```
op meta-pretty-print : Module Term -> QidList .
```

Esta operación toma como argumentos la representación de un módulo M y de un término t, y devuelve la lista de identificadores con comilla que codifican la cadena de tokens producida al imprimir t de forma edulcorada, utilizando la sintaxis concreta dada por M. Si se produce un error, se devuelve la lista vacía. Por ejemplo,

```
Maude> red meta-pretty-print(NAT, '_+_['s_[{'0}'Zero], 's_[{'0}'Zero]]) .
result QidList :  's '0 '+ 's '0
```

2.2.5. Estrategias internas

Como hemos visto, los módulos de sistema de Maude son teorías de reescritura que no tienen por qué ser confluentes ni terminantes. Por tanto, necesitamos controlar el proceso de reescritura, que en principio podría ir en muchas direcciones, mediante *estrategias* adecuadas. Gracias a las propiedades reflexivas de Maude, dichas estrategias pueden hacerse *internas* al sistema; es decir, pueden definirse en un módulo Maude, con el que se puede razonar como con cualquier otro módulo.

De hecho, hay una libertad absoluta para definir diferentes lenguajes de estrategias dentro de Maude, ya que los usuarios pueden definir sus propios lenguajes, sin estar limitados a un lenguaje fijo y cerrado. Para ello se utilizan las operaciones `meta-reduce`, `meta-apply` y `meta-rewrite` como expresiones de estrategias básicas, extendiéndose el módulo `META-LEVEL` con expresiones de estrategias adicionales y sus correspondientes reglas. A continuación presentaremos un ejemplo tomado de [CDE+99] que sigue la metodología para definir lenguajes de estrategias internos para lógicas reflexivas introducido en [Cla98].

Para ilustrar las ideas utilizaremos el módulo `SORTING` para ordenar vectores de enteros mostrado en la Sección 2.2.2. Veremos en concreto la forma en que puede controlarse la aplicación de las reglas.

Antes de explicar algunas de las estrategias que podemos definir, obsérvese que la estrategia por defecto del intérprete Maude para módulos de sistema puede ser utilizada, de forma sencilla y eficiente, por medio de la operación de descenso `meta-rewrite`:

```
Maude> rew meta-rewrite(SORTING,
             '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                 '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                 '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
             0) .
result Term: '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
                 '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
                 '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]]
```

Los lenguajes de estrategias pueden definirse en Maude en extensiones del módulo `META-LEVEL`. El módulo que definimos es el siguiente módulo `STRATEGY`, del que primero mostramos su sintaxis, para después introducir sus ecuaciones que serán ilustradas con ejemplos.

```
fmod STRATEGY is
  including META-LEVEL .
  sorts MetaVar Binding BindingList Strategy StrategyExpression .
  subsort MetaVar < Term .

  ops I J : -> MetaVar .
  op binding : MetaVar Term -> Binding .
  op nilBindingList : -> BindingList .
  op bindingList : Binding BindingList -> BindingList .

  op rewInWith : Module Term BindingList Strategy -> StrategyExpression .
  op set : MetaVar Term -> Strategy .
  op rewInWithAux : StrategyExpression Strategy -> StrategyExpression .
  op idle : -> Strategy .
  op failure : -> StrategyExpression .
  op and : Strategy Strategy -> Strategy .
  op apply : Qid -> Strategy .
  op applyWithSubst : Qid Substitution -> Strategy .
  op iterate : Strategy -> Strategy .
  op while : Term Strategy -> Strategy .
  op orelse : Strategy Strategy -> Strategy .

  op extTerm : ResultPair -> Term .
  op extSubst : ResultPair -> Substitution .
  op update : BindingList Binding -> BindingList .
  op applyBindingListSubst : Module Substitution BindingList -> Substitution .
  op substituteMetaVars : TermList BindingList -> TermList .

  op SORTING : -> Module .

  var M : Module .
  vars V V' F S G L : Qid .
  vars T T' : Term .
  var TL : TermList .
  var SB : Substitution .
  vars B B' : Binding .
  vars BL BL' : BindingList .
  vars MV MV' : MetaVar .
  vars ST ST' : Strategy .

  eq SORTING = SORTING .
```

En el módulo `STRATEGY` la operación `rewInWith` computa expresiones de estrategias. Los dos primeros argumentos son las metarrepresentaciones de un módulo M y un término t. El cuarto argumento es la estrategia S a computar, y el tercer argumento guarda

información que puede ser relevante para S, como veremos más adelante. La definición de `rewInWith` es tal que, al ir computando la estrategia, t se reescribe mediante una aplicación controlada de las reglas en M, se actualiza la información en el tercer argumento, y la estrategia S se reescribe a la estrategia que queda por ser computada. En caso de terminación, se acaba con la estrategia `idle`. La expresión de estrategia `failure` se devuelve si una estrategia no puede ser aplicada.

Una primera estrategia básica que podemos definir es la aplicación de una regla una vez y al nivel más alto de un término y con el primer encaje encontrado. Para esta estrategia básica se introduce el constructor `apply`, cuyo único argumento es la etiqueta de la regla a aplicar. La siguiente ecuación define el valor de `rewInWith` para esta estrategia:

```
eq rewInWith(M, T, BL, apply(L)) =
    if meta-apply(M, T, L, none, 0) == {error*, none}
    then failure
    else rewInWith(M, extTerm(meta-apply(M, T, L, none, 0)), BL, idle)
    fi .
```

Las operaciones `extTerm` y `extSubst` devuelven la primera y segunda componente, respectivamente, de un par construido con `{_,_}`.

```
eq extSubst({T, SB}) = SB .
eq extTerm({T, SB}) = T .
```

Podemos ilustrar el cómputo de una expresión de estrategia `apply` con el siguiente ejemplo:

```
Maude> rew rewInWith(SORTING,
              '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                  '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                  '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              nilBindingList,
              apply('sort)).
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'2}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'3}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'1}'NzMachineInt]],
          nilBindingList,
          idle)
```

La información relevante para el cómputo de una estrategia se almacena en una lista de ligaduras de valores a metavariables, donde los valores son de tipo `Term` y las metavariables se introducen por el usuario como constantes del tipo `MetaVar`.

El cómputo de la estrategia `set` actualiza la información almacenada asociando a una metavariable `MV` un término `T'`. Esto se lleva a cabo mediante la operación `update`. Antes se utiliza `substituteMetaVars` para sustituir las posibles metavariables en `T'` por su valor actual.

```
eq rewInWith(M, T, BL, set(MV, T')) =
   rewInWith(M, T,
       update(BL,
          binding(MV, meta-reduce(M, substituteMetaVars(T', BL)))),
       idle) .

eq substituteMetaVars(T, nilBindingList) = T .
eq substituteMetaVars(MV, bindingList(binding(MV', T'), BL)) =
   if MV == MV' then T' else substituteMetaVars(MV, BL) fi .
eq substituteMetaVars(F, BL) = F .
eq substituteMetaVars({F}S, BL) = {F}S .
eq substituteMetaVars(F[TL], BL) = F[substituteMetaVars(TL, BL)] .
eq substituteMetaVars((T, TL), BL) =
   (substituteMetaVars(T, BL), substituteMetaVars(TL, BL)).

eq update(bindingList(binding(MV, T), BL), binding(MV', T')) =
   if MV == MV' then bindingList(binding(MV, T'), BL)
               else bindingList(binding(MV, T),
                                update(BL, binding(MV', T')))
   fi .
eq update(nilBindingList, B) = bindingList(B, nilBindingList) .
```

El cómputo de la estrategia `applyWithSubst` aplica una regla, parcialmente instanciada con un conjunto de asignaciones, una vez al nivel más alto de un término, utilizando el primer encaje consistente con la sustitución parcial dada. Las representaciones de los términos asignados a las variables pueden contener metavariables que deben ser sustituidas por las representaciones a las que están ligadas en la lista actual de ligaduras. La operación `applyBindingListSubst` hace exactamente eso.

```
eq rewInWith(M, T, BL, applyWithSubst(L, SB)) =
   if meta-apply(M, T, L, applyBindingListSubst(M, SB, BL), 0)
          == {error*, none}
   then failure
   else rewInWith(M, extTerm(meta-apply(M, T, L,
                              applyBindingListSubst(M, SB, BL), 0)),
                  BL, idle)
   fi .

eq applyBindingListSubst(M, none, BL) = none .
eq applyBindingListSubst(M, ((V <- T); SB), BL) =
   ((V <- meta-reduce(M, substituteMetaVars(T, BL))) ;
    applyBindingListSubst(M, SB, BL)) .
```

Muchas de las estrategias interesantes se definen como concatenación o iteración de estrategias básicas. Para representar estos casos, extendemos el lenguaje de estrategias con los constructores `and`, `orelse`, `iterate` y `while`.

Las ecuaciones para las estrategias `and`, `orelse` e `iterate` se definen a continuación:

```
  eq rewInWith(M, T, BL, and(ST, ST')) =
      if rewInWith(M, T, BL, ST) == failure
      then failure
      else rewInWithAux(rewInWith(M, T, BL, ST), ST')
      fi .

  eq rewInWith(M, T, BL, orelse(ST, ST')) =
      if rewInWith(M, T, BL, ST) == failure
      then rewInWith(M, T, BL, ST')
      else rewInWith(M, T, BL, ST)
      fi .

  eq rewInWith(M, T, BL, iterate(ST)) =
      if rewInWith(M, T, BL, ST) == failure
      then rewInWith(M, T, BL, idle)
      else rewInWithAux(rewInWith(M, T, BL, ST), iterate(ST))
      fi .
```

donde la operación `rewInWithAux` se define mediante la ecuación

```
  eq rewInWithAux(rewInWith(M, T, BL, idle), ST) = rewInWith(M, T, BL, ST) .
```

lo que fuerza que el cómputo de una secuencia de estrategias se realice paso a paso, en el sentido de que una estrategia solo se considerará cuando la anterior ya haya terminado completamente.

Podemos ilustrar el cómputo de estas estrategias con los siguientes ejemplos:

```
Maude> rew rewInWith(SORTING,
            '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              nilBindingList,
              and(set(I, {'3}'MachineInt),
                  applyWithSubst('sort, ('I <- I)))) .
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]],
          bindingList(binding(I, {'3}'NzMachineInt), nilBindingList),
          idle)

Maude> rew rewInWith(SORTING,
            '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
                '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
                '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              bindingList(binding(J, {'2}'MachineInt), nilBindingList),
      orelse(applyWithSubst('sort, ('J <- {'4}'MachineInt)),
             applyWithSubst('sort, ('J <- J)))).
```

```
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'3}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'1}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'2}'NzMachineInt]],
          bindingList(binding(J, {'2}'MachineInt), nilBindingList),
            idle)

Maude> rew rewInWith(SORTING,
           '__['<_;_>[{'1}'MachineInt, {'3}'MachineInt],
               '<_;_>[{'2}'MachineInt, {'2}'MachineInt],
               '<_;_>[{'3}'MachineInt, {'1}'MachineInt]],
              nilBindingList,
              iterate(apply('sort))).
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt]],
          nilBindingList, idle)
```

Finalmente, la estrategia `while` hace que el cómputo de una estrategia dada dependa de que se cumpla una condición. Esta condición debería ser la representación de un término de tipo `Bool`.

```
  eq rewInWith(M, T, BL, while(T', ST)) =
      if meta-reduce(M, substituteMetaVars(T', BL)) == {'true}'Bool
      then (if rewInWith(M, T, BL, ST) == failure
            then rewInWith(M, T, BL, idle)
            else rewInWithAux(rewInWith(M, T, BL, ST), while(T', ST))
            fi)
      else rewInWith(M, T, BL, idle)
      fi .
```

El lenguaje de estrategias anterior se puede extender para definir, por ejemplo, el algoritmo de *ordenación por inserción*. La siguiente estrategia `insertion-sort`(n) puede utilizarse para ordenar un vector de enteros de longitud n.

```
  op insertion-sort : MachineInt -> Strategy .
  ops X Y : -> MetaVar .

  var N : MachineInt .

  eq insertion-sort(N) =
     and(set(Y, {'2}'MachineInt),
     while('_<=_[Y, {index(', N)}'MachineInt],
           and(set(X, Y),
           and(while('_>_[X, {'1}'MachineInt],
                     and(applyWithSubst('sort,
```

```
                          (('I <- X);
                          ('J <- '_-_[X, {'1}'MachineInt]))),
                    set(X, '_-_[X, {'1}'MachineInt]))),
          set(Y, '_+_[Y, {'1}'MachineInt])))))) .
```

Por ejemplo, podemos utilizar la estrategia `insertion-sort` para ordenar un vector de enteros de longitud 10:

```
Maude> rew rewInWith(SORTING,
              '__['<_;_>[{'1}'MachineInt, {'10}'MachineInt],
                  '<_;_>[{'2}'MachineInt, {'9}'MachineInt],
                  '<_;_>[{'3}'MachineInt, {'8}'MachineInt],
                  '<_;_>[{'4}'MachineInt, {'7}'MachineInt],
                  '<_;_>[{'5}'MachineInt, {'6}'MachineInt],
                  '<_;_>[{'6}'MachineInt, {'5}'MachineInt],
                  '<_;_>[{'7}'MachineInt, {'4}'MachineInt],
                  '<_;_>[{'8}'MachineInt, {'3}'MachineInt],
                  '<_;_>[{'9}'MachineInt, {'2}'MachineInt],
                  '<_;_>[{'10}'MachineInt, {'1}'MachineInt]],
               nilBindingList,
               insertion-sort(10)) .
result StrategyExpression:
rewInWith(SORTING,
          '__['<_;_>[{'1}'NzMachineInt,{'1}'NzMachineInt],
              '<_;_>[{'2}'NzMachineInt,{'2}'NzMachineInt],
              '<_;_>[{'3}'NzMachineInt,{'3}'NzMachineInt],
              '<_;_>[{'4}'NzMachineInt,{'4}'NzMachineInt],
              '<_;_>[{'5}'NzMachineInt,{'5}'NzMachineInt],
              '<_;_>[{'6}'NzMachineInt,{'6}'NzMachineInt],
              '<_;_>[{'7}'NzMachineInt,{'7}'NzMachineInt],
              '<_;_>[{'8}'NzMachineInt,{'8}'NzMachineInt],
              '<_;_>[{'9}'NzMachineInt,{'9}'NzMachineInt],
              '<_;_>[{'10}'NzMachineInt,{'10}'NzMachineInt]],
          bindingList(binding(Y, {'11}'NzMachineInt),
          bindingList(binding(X, {'1}'NzMachineInt), nilBindingList)),
       idle)
```

En [CDE+99] se extiente este lenguaje de estrategias para definir una estrategia ganadora de una versión del juego de Nim, y para definir un metaintérprete de módulos Maude que solo contengan reglas confluentes y terminantes. En [CDE+02, Sección 6] se utiliza una variación del módulo `SORTING` para ilustrar cómo se puede controlar el proceso de ordenación de un vector utilizando unas estrategias diferentes a las que hemos presentado aquí.

En el Capítulo 4 definiremos nuestra propia estrategia para controlar la aplicación de las reglas semánticas del álgebra de procesos CCS, y en el Capítulo 8 definiremos una estrategia para poder demostrar la corrección del protocolo de elección de líder del estándar IEEE 1394.

2.2.6. El módulo predefinido LOOP-MODE

Utilizando conceptos orientados a objetos, se ha especificado en Maude un mecanismo general de entrada/salida proporcionado por el módulo predefinido `LOOP-MODE` que extiende el módulo `QID-LIST` con un bucle de lectura, evaluación y escritura [CDE+99, Dur99].

```
mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System .
endm
```

El operador `[_,_,_]` puede verse como un *objeto persistente* con un canal de entrada (su primer argumento), un canal de salida (su tercer argumento) y un estado (su segundo argumento). Además de tener canales de entrada y salida, los términos de tipo `System` ofrecen la posibilidad de mantener un estado persistente en su segunda componente. Este estado ha sido declarado de una forma completamente genérica. De hecho, el tipo `State` del módulo `LOOP-MODE` no tiene ningún constructor definido. Esto da completa flexibilidad a la hora de definir los términos que se quiere que representen el estado persistente del bucle en una aplicación particular. El tratamiento de la entrada y la salida, y de la evolución del estado interno se definen mediante reglas de reescritura apropiadas.

Podemos ilustrar estas ideas con un ejemplo "de juguete", en el que el bucle de entrada/salida muestra en la salida la entrada, pero solo después de que se hayan introducido diez tokens, es decir, mantiene la entrada hasta que el número de tokens almacenados en el estado sea diez. Es necesario un estado persistente que almacene la entrada ya introducida y no mostrada en la salida. Este estado puede representarse por un par formado por una lista de identificadores con comilla (la entrada desde la última salida) y un número que mida la longitud de la lista.

```
mod DUPLICATE-TEN is
  including LOOP-MODE .
  protecting MACHINE-INT .

  op <_;_> : QidList MachineInt -> State .
  op init : -> System .

  vars Input StoredInput Output : QidList .
  vars QI QI0 QI1 QI2 QI3 QI4 QI5 QI6 QI7 QI8 QI9 : Qid .
  var Counter : MachineInt .

  rl [init] : init
           => [nil, < nil ; 0 >, nil] .
  rl [in] : [QI Input, < StoredInput ; Counter >, Output]
         => [Input, < StoredInput QI ; Counter + 1 >, Output] .
  rl [out] : [Input,
              < QI0 QI1 QI2 QI3 QI4 QI5 QI6 QI7 QI8 QI9 StoredInput ;
                Counter >,
              Output]
```

```
          => [Input,
              < StoredInput ; Counter - 10 >,
              Output QI0 QI1 QI2 QI3 QI4 QI5 QI6 QI7 QI8 QI9] .
endm
```

Una vez introducido este módulo, se tiene que inicializar el bucle utilizando el comando `loop`. Para distinguir la entrada dirigida directamente al sistema Maude de la entrada dirigida al bucle de entrada/salida, esta última se encierra entre paréntesis. Cuando se escribe algo entre paréntesis se convierte en una lista de identificadores con comilla y se coloca en la primera componente del bucle. La salida se trata de forma inversa, es decir, la lista de identificadores con comilla colocada en la tercera componente del bucle se muestra en el terminal aplicando la conversión inversa.

```
Maude> loop init .

Maude> (a b)

Maude> (c d e f g h i)

Maude> (j k l)

a b c d e f g h i j
```

En la Sección 7.6.5 utilizaremos estas ideas para implementar una herramienta interactiva en la que ejecutar procesos de Full LOTOS.

2.3. Full Maude

Full Maude es una extensión del lenguaje Maude, escrita en el propio (Core) Maude, con notación para la programación orientada a objetos, módulos parametrizados, vistas (para la instanciación de módulos) y expresiones de módulos [Dur99]. Full Maude admite además los módulos funcionales y de sistema de Maude, y muchos de los comandos utilizados para trabajar con estos módulos. A continuación nos centraremos en las características adicionales que ofrece Full Maude.

2.3.1. Módulos orientados a objetos

En un sistema concurrente orientado a objetos el estado, denominado usualmente *configuración*, tiene la estructura de un multiconjunto formado por objetos y mensajes que evoluciona por reescritura módulo asociatividad, conmutatividad e identidad, utilizando reglas de reescritura que describen los efectos de los *eventos de comunicación* entre objetos y mensajes. Por tanto, se puede ver el cómputo concurrente orientado a objetos como la deducción en la lógica de reescritura. Las configuraciones S alcanzables a partir de una configuración inicial S_0 son exactamente aquellas tales que el secuente $S_0 \longrightarrow S$ se puede probar en lógica de reescritura utilizando las reglas que especifican el comportamiento del sistema orientado a objetos.

Un *objeto* con un cierto estado se representa como un término

$$< O : C \mid a_1 : v_1 , \ \ldots , \ a_n : v_n >$$

donde O es el nombre del objeto, C es el identificador de su clase, los a_i son los identificadores de los atributos del objeto, y los v_i son los correspondientes valores. Los *mensajes* no tienen una sintaxis fija, sino que esta la define el usuario para cada aplicación. El estado concurrente de un sistema orientado a objetos es un multiconjunto de objetos y mensajes, de tipo `Configuration`, con el operador (de sintaxis vacía) `__` como operador de unión de multiconjuntos. El siguiente módulo, importado implícitamente por cualquier módulo orientado a objetos, define los conceptos básicos de un sistema orientado a objetos.

```
fmod CONFIGURATION is
  sorts Oid Cid Attribute AttributeSet Object Msg Configuration .
  subsorts Object Msg < Configuration .
  subsort Attribute < AttributeSet .

  op none : -> AttributeSet .
  op _,_ : AttributeSet AttributeSet -> AttributeSet [assoc comm id: none] .

  op <_:_| > : Oid Cid -> Object .
  op <_:_|_> : Oid Cid AttributeSet -> Object .

  op none : -> Configuration .
  op __ : Configuration Configuration -> Configuration [assoc comm id: none] .
endfm
```

En Full Maude los sistemas concurrentes orientados a objetos se describen por medio de *módulos orientados a objetos* (introducidos con las palabras clave `omod...endom`) con una sintaxis especial más conveniente para estos sistemas. Por ejemplo, el módulo orientado a objetos `ACCNT` especifica el comportamiento concurrente de los objetos de una clase muy sencilla `Accnt` de cuentas bancarias, con un atributo `bal` con el saldo de la cuenta, y que pueden recibir mensajes para aumentar y decrementar el saldo, o para hacer transferencias entre dos cuentas[6].

```
(omod ACCNT is
  protecting QID .
  protecting MACHINE-INT .

  subsort Qid < Oid .

  class Accnt | bal : MachineInt .

  msgs credit debit : Oid MachineInt -> Msg .
  msg transfer_from_to_ : MachineInt Oid Oid -> Msg .
```

[6]Full Maude recibe la entrada a través del objeto persistente del módulo `LOOP-MODE` descrito en la Sección 2.2.6, por lo que esta debe escribirse encerrada entre paréntesis.

```
vars A B : Oid .
vars M N N' : MachineInt .

rl [credit] : credit(A, M) < A : Accnt | bal : N >
           => < A : Accnt | bal : (N + M) > .
crl [debit] : debit(A, M) < A : Accnt | bal : N >
           => < A : Accnt | bal : (N - M) >
           if N > M .
crl [transfer] : (transfer M from A to B)
                   < A : Accnt | bal : N > < B : Accnt | bal : N' >
              => < A : Accnt | bal : (N - M) > < B : Accnt | bal : (N' + M) >
               if N > M .
endom)
```

Las clases se definen con la palabra clave `class`, seguida del nombre de la clase y de la lista de declaraciones de atributos separados por coma. Cada declaración de atributo es de la forma $a : S$, donde a es el identificador del atributo y S el tipo de los valores que este atributo puede tomar. Es decir, una declaración de clase tiene en general la forma:

$$\texttt{class}\ C\ \texttt{|}\ a_1\texttt{:}\ S_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{:}\ S_n\ \texttt{.}$$

Los mensajes se introducen con la palabra clave `msg` (o `msgs`) y tienen una sintaxis definida por el usuario.

Las reglas en un módulo orientado a objetos especifican de forma declarativa el comportamiento asociado a los mensajes. La estructura como multiconjunto de la configuración proporciona la estructura distribuida al nivel más externo del sistema, y permite la aplicación concurrente de las reglas [Mes93]. La forma general de tales reglas es la siguiente:

$$\begin{aligned} & M_1 \ldots M_n\ \langle O_1 : F_1 \mid atts_1\rangle \ldots \langle O_m : F_m \mid atts_m\rangle \\ \longrightarrow\ & \langle O_{i_1} : F'_{i_1} \mid atts'_{i_1}\rangle \ldots \langle O_{i_k} : F'_{i_k} \mid atts'_{i_k}\rangle \\ & \langle Q_1 : D_1 \mid atts''_1\rangle \ldots \langle Q_p : D_p \mid atts''_p\rangle \\ & M'_1 \ldots M'_q \\ & \textit{if } C \end{aligned}$$

donde $k, p, q \geq 0$, los M_s son mensajes, los $i_1, \ldots, i_k$ son números diferentes entre los originales $1, \ldots, m$, y C es la condición de la regla. El resultado de la aplicación de una regla tal es la desaparición de los mensajes $M_1, \ldots, M_n$; el estado y, posiblemente, la clase de los objetos $O_{i_1}, \ldots, O_{i_k}$ pueden cambiar; el resto de objetos O_j desaparecen; se crean los nuevos objetos $Q_1, \ldots, Q_p$; y se envían nuevos mensajes $M'_1, \ldots, M'_q$.

Ya que en esta última regla aparecen varios objetos y mensajes en el lado izquierdo, se dice que es una *regla síncrona*. Es importante conceptualmente distinguir el caso especial de las reglas que tienen como mucho un objeto y un mensaje en su lado izquierdo. Estas reglas se denominan *reglas asíncronas* y tienen la forma

$$
\begin{aligned}
&(M)\ \langle O : F \mid atts \rangle \\
&\longrightarrow (\langle O : F' \mid atts' \rangle) \\
&\quad \langle Q_1 : D_1 \mid atts''_1 \rangle \ldots \langle Q_p : D_p \mid atts''_p \rangle \\
&\quad M'_1 \ldots M'_q \\
&\ \textit{if}\ C
\end{aligned}
$$

donde aparecen entre paréntesis los elementos opcionales.

Por convenio, en una regla solo se mencionan explícitamente los atributos de un objeto relevantes para ella. En particular, los atributos que solo se mencionan en el lado izquierdo de la regla permanecen sin modificación, el valor original de los atributos mencionados solo en el lado derecho de la regla carece de importancia, y todos los atributos no mencionados explícitamente permanecen sin modificar.

En el ejemplo anterior, podemos reescribir una configuración muy simple formada por una cuenta y un mensaje de la siguiente manera:

```
Maude> (rew < 'Peter : Accnt | bal : 2000 > debit('Peter, 1000) .)
result Object : < 'Peter : Accnt | bal : 1000 >
```

La herencia entre clases viene soportada directamente por la estructura de tipos ordenados de Maude. Una declaración de subclase `C < C'` (introducida con la palabra clave `subclass`) en un módulo orientado a objetos es un caso particular de una declaración de subtipos `C < C'`. El efecto de la declaración de subclase es que los atributos, mensajes y reglas de todas las superclases así como los nuevos atributos, mensajes y reglas definidos en la subclase caracterizan la estructura y el comportamiento de los objetos de la subclase.

Por ejemplo, podemos definir una clase de cuentas de ahorro introduciendo una subclase `SavAccnt` de `Accnt` con un nuevo atributo `rate` que almacena el interés de la cuenta.

```
(omod SAV-ACCNT is
  including ACCNT .

  class SavAccnt | rate : MachineInt .
  subclass SavAccnt < Accnt .
 endom)
```

Aunque los módulos orientados a objetos proporcionan una sintaxis conveniente para la programación de sistemas orientados a objetos, su semántica puede reducirse a la de los módulos de sistema. De hecho, cada módulo orientado a objetos puede traducirse a un módulo de sistema correspondiente cuya semántica es por definición la del módulo original orientado a objetos. En [Dur99, CDE+99] se da una explicación detallada de esta traducción, que Full Maude realiza cuando se introducen en el sistema módulos orientados a objetos.

2.3.2. Programación parametrizada

Los componentes básicos de la programación parametrizada son los módulos parametrizados, las teorías y las vistas.

Las teorías se utilizan para declarar interfaces de módulos, es decir, las propiedades sintácticas y semánticas que tienen que satisfacer los módulos utilizados como parámetros actuales en una instanciación. Full Maude soporta tres tipos de teorías: funcionales (`fth...endfth`), de sistema (`th...endth`) y orientadas a objetos (`oth...endoth`). Su estructura es la misma que la del tipo de módulo correspondiente. Las teorías son teorías de lógica de reescritura con semántica laxa, es decir, cualquier modelo que satisfaga los axiomas es admisible [Mes92].

La siguiente teoría funcional `TRIV` requiere simplemente la presencia de un tipo.

```
(fth TRIV is
  sort Elt .
 endfth)
```

La teoría de los conjuntos parcialmente ordenados con un operador binario transitivo y antireflexivo puede expresarse de la siguiente manera:

```
(fth POSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false .
  ceq X < Z = true if X < Y and Y < Z .
 endfth)
```

Las teorías pueden utilizarse para declarar los requisitos del interfaz de los módulos parametrizados. Estos pueden parametrizarse con una o más teorías. Todas las teorías tienen que estar etiquetadas de forma que sus tipos puedan ser identificados de manera única. Además, en la versión 1.0.5 de Full Maude todos los tipos importados de las teorías en un interfaz tienen que ser cualificados con sus etiquetas: si Z es la etiqueta de una teoría parámetro T, cada tipo S de T tiene que ser cualificado como $S\,.\,Z$.

El módulo parametrizado `SET` que define los conjuntos finitos, con `TRIV` como interfaz, puede definirse de la siguiente manera:

```
(fmod SET[X :: TRIV] is
  sorts Set NeSet .
  subsorts Elt.X < NeSet < Set .
  op mt : -> Set .
  op __ : Set Set -> Set [assoc comm id: mt] .
  op __ : NeSet NeSet -> NeSet [assoc comm id: mt] .
  var E : Elt.X .
  eq E E = E .
 endfm)
```

Las vistas se utilizan para indicar cómo un módulo o teoría destino particular satisface una teoría fuente. En la versión 1.0.5 de Full Maude, todas las vistas tienen que ser definidas explícitamente, y todas tienen que tener un nombre. En la definición de una vista hay que indicar su nombre, la teoría fuente, el módulo o teoría destino y la correspondencia de cada tipo, operación, clase o mensaje.

La siguiente vista

```
(view MachineInt from TRIV to MACHINE-INT is
   sort Elt to MachineInt .
 endv)
```

define una vista desde la teoría `TRIV` al módulo `MACHINE-INT`.

La instanciación es el proceso por el cual se asignan parámetros actuales a los parámetros de un módulo parametrizado, creando como resultado un nuevo módulo. La instanciación requiere una vista desde cada parámetro formal al correspondiente parámetro actual. La instanciación de un módulo parametrizado tiene que hacerse con vistas definidas explícitamente con anterioridad. Por ejemplo, podemos obtener los conjuntos de enteros con la expresión de módulo `SET[MachineInt]`.

2.3.3. Extensiones de `META-LEVEL`

En Full Maude, dentro de cualquier módulo que incluya `META-LEVEL` se puede utilizar la función `up` para obtener la metarrepresentación de un término dentro de un módulo o la metarrepresentación de un módulo. Por ejemplo, para obtener la metarrepresentación del término `s 0` en el módulo `NAT` (Sección 2.2.4), podemos escribir

```
Maude> (red up(NAT, s 0) .)
result Term : 's_[{'0}'Zero]
```

La función `up` también nos permite acceder a la metarrepresentación de un módulo ya introducido en el sistema. Evaluando en cualquier módulo que incluya `META-LEVEL` la función `up` con el nombre de un módulo en el sistema se obtiene la metarrepresentación de dicho módulo. Por ejemplo, podemos obtener la metarrepresentación del módulo `NAT` de la siguiente manera:

```
Maude> (red up(NAT) .)
result FModule :
      fmod 'NAT is
        nil
        sorts 'Zero ; 'Nat .
        subsort 'Zero < 'Nat .
        op '0 : nil -> 'Zero [none] .
        op 's_ : 'Nat -> 'Nat [none] .
        op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
        op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
        var 'N : 'Nat .
```

```
    var 'M : 'Nat .
    none
    eq '_+_[{'0}'Zero, 'N] = 'N .
    eq '_+_['s_['N], 'M] = 's_['_+_['N, 'M]] .
    eq '_*_[{'0}'Zero, 'N] = {'0}'Zero .
    eq '_*_['s_['N], 'M] = '_+_['M, '_*_['N, 'M]] .
endfm
```

Esta operación puede utilizarse para escribir reducciones como las presentadas en la Sección 2.2.4. Por ejemplo, la reducción `meta-reduce(NAT, s 0 + s 0)` puede escribirse de la siguiente manera:

```
Maude> (red meta-reduce(up(NAT), up(NAT, s 0 + s 0)) .)
result Term : 's_['s_[{'0}'Zero]]
```

El resultado de un cómputo al metanivel, utilizando eventualmente varios niveles de reflexión, puede ser un término o un módulo metarrepresentado una o más veces, que puede ser difícil de leer. Para mostrar la salida de una forma más legible se puede utilizar el comando `down`, que es, en cierta forma, el inverso de `up`, ya que construye un término a partir de su metarrepresentación. El comando `down` recibe dos argumentos: el nombre del módulo al cual pertenece el término t que tiene que devolver y el comando cuyo resultado es la metarrepresentación del término t. Por ejemplo, se puede escribir

```
Maude> (down NAT :
          red meta-reduce(up(NAT), up(NAT, s 0 + s 0)) .)
result Nat : s s 0
```

2.3.4. Restricciones en la sintaxis de Full Maude

En Full Maude podemos escribir módulos funcionales y de sistema como en Core Maude (Maude básico sin extensiones), pero encerrándolos entre paréntesis. Sin embargo, hay algunas diferencias entre lo que permiten Core Maude y Full Maude. Al escribir especificaciones en Full Maude hay que seguir las siguientes restricciones:

1. Los nombres de operadores y mensajes, cuando son declarados, deben darse en su *forma de identificador único* equivalente.

2. Los nombres de tipos utilizados en la cualificación de términos y en los axiomas de pertenencia deben darse en su *forma de identificador único* equivalente.

Así, por ejemplo, el operador `_menor o igual que_` tiene que ser declarado en Full Maude como `_menor‘o‘igual‘que_`. Excepto por esto, la declaración de operadores es igual que en Core Maude:

```
op _menor‘o‘igual‘que_ : Nat Nat -> Bool .
```

Téngase en cuenta que no solo los espacios en blanco, sino también los caracteres especiales '`{`', '`}`', '`(`', '`)`', '`[`', '`]`' y '`,`' rompen los identificadores. Por tanto, para declarar en Full Maude un operador como `{_}` que tome un argumento de tipo `Nat` y construya un valor de tipo `Set`, deberemos escribir

```
op `{_`} : Nat -> Set .
```

Ya que la cualificación de términos con tipos o los axiomas de pertenencia son tratados directamente por Core Maude, que no conoce los tipos parametrizados, el usuario debe utilizar en estos casos los nombres de los tipos parametrizados, no como los ha definido, sino en su forma equivalente de identificador único. Por tanto, si tenemos por ejemplo un tipo `List[Nat]` y una constante `nil`, si es necesario debería cualificarse como `(nil).List`[`Nat`].

2.4. Maude 2.0

Maude 2.0 es la nueva versión de Maude actualmente en desarrollo cuyas principales características son: una mayor generalidad y expresividad; un soporte eficiente de un mayor número de aplicaciones de programación; y su utilidad como una componente principal en el desarrollo de programación en internet y de sistemas de computación móviles [CDE+00b]. En esta sección resumiremos brevemente las nuevas características de Maude 2.0 utilizadas en los capítulos siguientes[7], y que consisten en:

- soporte más amplio de la lógica ecuacional de pertenencia,
- nueva sintaxis en la declaración de variables,
- reglas con reescritura en las condiciones,
- comandos de búsqueda,
- más atributos para declarar propiedades de los operadores,
- nuevos tipos predefinidos, y
- nueva sintaxis para el metanivel.

La nueva sintaxis de módulos funcionales extiende la sintaxis previa para permitir la máxima generalidad posible en el soporte de especificaciones de la lógica ecuacional de pertenencia. Para dar soporte a la especificación de operaciones parciales que no se restringen a una función total sobre un producto de tipos, se permite la declaración explícita de tales funciones al nivel de las *familias* (*kinds* en inglés). En la lógica ecuacional de pertenencia los términos que no tienen tipo se entiende que son términos *no definidos* o *error*. Una familia k tiene un conjunto S_k de *tipos* asociados, que se entienden semánticamente como

[7] En el momento de escribir esta tesis, existen ya implementaciones de Maude 2.0 en fase de prueba, no distribuibles al público en general.

subconjuntos suyos. En general, una función total al nivel de las familias se restringe a una función parcial al nivel de los tipos.

En los módulos funcionales, las familias no se nombran explícitamente. En cambio, se identifica una familia k con el conjunto S_k de sus tipos, entendido como una clase de equivalencia módulo la relación de equivalencia generada por la ordenación de subtipado. Por tanto, para cualquier $s \in S_k$, $[s]$ denota la familia k, entendida como la componente conexa a la que pertenece el tipo s.

Considérese, por ejemplo, la operación de concatenación de caminos de un grafo en el módulo `PATH` de la Sección 2.2.1. Realmente se trata de una función parcial, aunque allí se definió como total sobre el tipo `Path?` de caminos confusos. Resulta más simple, y más elegante, definirla al nivel de las familias mediante la declaración

```
op _;_ : [Path] [Path] -> [Path] .
```

El módulo `PATH` completo en Maude 2.0 es el siguiente

```
fmod PATH is
  protecting NAT .

  sorts Edge Path Node .
  subsort Edge < Path .

  ops n1 n2 n3 n4 n5 : -> Node .
  ops a b c d e f : -> Edge .
  op _;_ : [Path] [Path] -> [Path] .
  ops source target : Path -> Node .
  op length : Path -> Nat .

  var E : Edge .
  vars P Q : Path .

  cmb (E ; P) : Path if target(E) == source(P) .

  ceq source(P) = source(E) if E ; Q := P .
  ceq target(P) = target(P) if E ; Q := P .
  eq length(E) = 1 .
  ceq length(E ; P) = 1 + length(P) if E ; P : Path .

  eq source(a) = n1 .    eq target(a) = n2 .
  eq source(b) = n1 .    eq target(b) = n3 .
  eq source(c) = n3 .    eq target(c) = n4 .
  eq source(d) = n4 .    eq target(d) = n2 .
  eq source(e) = n2 .    eq target(e) = n5 .
  eq source(f) = n2 .    eq target(f) = n1 .
endfm
```

En Maude 2.0 una variable es un identificador formado por un nombre y un tipo, separados por ':'. Por ejemplo, `N:Nat` es una variable de tipo `Nat`. De esta forma no es

necesario declarar las variables, sino que pueden aparecer directamente en los términos. Las declaraciones de variables, sin embargo, se siguen permitiendo por conveniencia.

Las reglas de reescritura pueden tomar la forma más general posible de la variante de la lógica de reescritura construida sobre la lógica ecuacional de pertenencia. Es decir, las reglas pueden ser de la forma

$$t \longrightarrow t' \;\; \textit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

sin ninguna restricción sobre las variables nuevas que puedan aparecer en la parte derecha o en la condición. Las condiciones de las reglas se construyen por medio de una conectiva de conjunción asociativa `/\`, que permite unir ecuaciones (tanto ecuaciones ordinarias `t = t'`, como ecuaciones de encaje `t := t'`), axiomas de pertenencia (`t : s`), y reescrituras (`t => t'`) como condiciones. En esta total generalización la ejecución de un módulo de sistema requerirá *estrategias* que controlen al metanivel la instanciación de las variables nuevas en la condición y en la parte derecha. Sin embargo, el intérprete por defecto de Maude 2.0 es capaz de ejecutar por sí mismo una clase de módulos de sistema bastante general, denominados *módulos admisibles*. Esencialmente, el requerimiento de admisibilidad asegura que todas las variables nuevas llegarán a ser instanciadas por encaje de patrones [CDE+00b].

Cuando se ejecuta una regla condicional, la comprobación de satisfacción de todas sus condiciones se lleva a cabo secuencialmente de izquierda a derecha; pero obsérvese que además del hecho de que pueden ser posibles diferentes encajes en las condiciones ecuacionales debido a la presencia de axiomas estructurales, también hay que tratar el hecho de que la resolución de las condiciones de reescritura requiere una *búsqueda*, incluyendo la posibilidad de búsqueda de nuevas soluciones cuando las encontradas anteriormente no han satisfecho las condiciones siguientes. Por tanto, el intérprete por defecto debe soportar cómputos de búsqueda. El comando `search` busca todas las reescrituras de un término dado que encajan con un patrón dado satisfaciendo alguna condición.

La sintaxis de este comando es la siguiente:

`search` t *search-type* P `such that` C `.`

donde t es un término, P es un patrón, C es una condición, y el tipo de búsqueda *search-type* puede ser:

- `=>`, exactamente una reescritura;
- `=>*`, cero o más reescrituras;
- `=>+`, una o más reescrituras;
- `=>!`, hasta que no se apliquen más reglas.

Otra característica de Maude 2.0 es el atributo `frozen`. Cuando un operador se declara como *congelado*, sus argumentos no pueden ser reescritos por reglas. En la Sección 5.2 se

explicará con todo detalle por qué este atributo es útil, e incluso imprescindible en algunas ocasiones. Nótese que la utilización de este atributo cambia la semántica del operador congelado, al no permitir la regla de congruencia de la lógica de reescritura. Aunque en la implementación actual de Maude 2.0 que nosotros vamos a utilizar, el atributo `frozen` congela todos los argumentos de un operador, ya se está pensando en generalizar este atributo para que se pueda indicar qué argumentos son los congelados.

Maude 2.0 ofrece una colección más amplia que Maude 1.0.5 de módulos predefinidos. Existen módulos predefinidos que implementan una jerarquía de valores númericos: naturales (`NAT`), enteros (`INT`), racionales (`RAT`) y números reales con coma flotante (`FLOAT`). Todos ellos con las operaciones típicas y con implementaciones muy eficientes. Existe también un módulo que implementa las cadenas de caracteres (`STRING`) con sus operaciones típicas y un módulo que implementa conversiones entre los valores numéricos y las cadenas (`NUMBER-CONVERSION`).

2.4.1. `META-LEVEL` en Maude 2.0

El nuevo módulo `META-LEVEL`, además de permitir la metarrepresentación de la sintaxis extendida, ofrece una representación más simple de los términos y un conjunto más rico de funciones de descenso.

La representación más simple de los términos se obtiene a través de los subtipos `Constant` y `Variable` del tipo `Qid`. Las constantes metarrepresentadas son identificadores con comilla que contienen el nombre de la constante y su tipo separados por un '`.`', por ejemplo, `'0.Zero`. De forma similar, las variables metarrepresentadas contienen su nombre y su tipo separados por '`:`', por ejemplo, `'N:Nat`.

Los términos se construyen entonces de la forma habitual, aplicando un símbolo de operador a una lista de términos, siendo los casos básicos de la construcción las constantes y las variables.

```
subsorts Constant Variable < Term .
op _[_] : Qid TermList -> Term .

subsort Term < TermList .
op _,_ : TermList TermList -> TermList [assoc] .
```

Por ejemplo, el término `s 0 + N:Nat` en un módulo `NAT` se metarrepresenta ahora por

```
'_+_['s_['0.Zero],'N:Nat].
```

Los módulos se metarrepresentan prácticamente de la misma manera que en la versión anterior de Maude, con las siguientes diferencias: se necesita sintaxis nueva para las nuevas condiciones; no hay necesidad de declaración de variables; y los términos utilizan la nueva metarrepresentación de términos.

Por ejemplo, el módulo `NAT`

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
endfm
```

se metarrepresenta ahora de la siguiente manera:

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
  op 's : 'Nat -> 'Nat [none] .
  op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
  none
  eq '_+_['0.Zero, 'N:Nat] = 'N:Nat .
  eq '_+_['s['N:Nat], 'M:Nat] = 's['_+_['N:Nat, 'M:Nat]] .
endfm
```

En Maude 2.0 ha cambiado la sintaxis de las funciones de descenso, y el tipo del resultado que devuelven, que ahora incluye también el tipo del término resultado devuelto.

```
  sort Bound .
  subsort Nat < Bound .
  op unbounded : -> Bound .

  op metaReduce : Module Term -> [ResultPair] .
  op metaRewrite : Module Term Bound -> [ResultPair] .
  op metaApply : Module Term Qid Substitution Nat -> [ResultTriple] .
  op {_,_} : Term Type -> ResultPair .
  op {_,_,_} : Term Type Substitution -> ResultTriple .
  op failure : -> [ResultTriple] .
```

Además se ha añadido, entre otras, una función que representa al metanivel la búsqueda realizada por el comando `search`.

```
  op metaSearch : Module Term Term Condition Qid Bound Nat -> [ResultTriple] .
```

La función `metaSearch` recibe como argumentos la metarrepresentación del módulo con el que se quiere trabajar, el término inicial para la búsqueda, el patrón que hay que

buscar, una condición adicional, la clase de búsqueda (`'*` para cero o más reescrituras, `'+` para una o más reescrituras, y `'!` para emparejar solo con términos que no pueden ser reescritos más), la profundidad de la búsqueda y el número de solución requerida, y devuelve el término que encaja con el patrón, su tipo y la sustitución producida por el emparejamiento.

La sintaxis de la operación que realiza el análisis sintáctico de términos en un módulo, y la de la operación encargada de la impresión edulcorada de un término metarrepresentado también se han modificado.

```
op metaParse : Module QidList Type? -> [ResultPair] .
op metaPrettyPrint : Module Term -> QidList .
```

El tercer argumento de `metaParse` indica el tipo en el que se tiene que intentar hacer el análisis sintáctico, pudiéndose utilizar la constante `anyType` para representar un tipo cualquiera.

Capítulo 3

Semánticas operacionales ejecutables

Como ya se ha dicho previamente, la lógica de reescritura fue introducida por José Meseguer para servir como modelo unificado de la concurrencia en el cual diversos modelos de concurrencia bien conocidos y estudiados pudieran ser representados en un único marco común [Mes92]. Este objetivo fue extendido en [MOM93] presentando la lógica de reescritura como un *marco semántico* en el cual se pudieran representar de forma natural muchos modelos de computación (particularmente concurrentes y distribuidos) y lenguajes diferentes. Debido a la dualidad intrínseca entre lógica y computación que ofrece la lógica de reescritura, las mismas razones que hacen de ella un marco semántico muy apropiado, hacen también que sea un *marco lógico* muy atractivo en el cual representar gran variedad de lógicas diferentes. En [MOM93] se demostró que varias lógicas de naturaleza muy diferente pueden representarse dentro de la lógica de reescritura de una forma directa y natural. La forma general de conseguir dichas representaciones consiste en:

- Representar las fórmulas, o más en general las estructuras de demostración tales como secuentes, como términos de un tipo de datos en una lógica ecuacional, cuyas ecuaciones expresen axiomas estructurales naturales a la lógica en cuestión.

- Representar las reglas de inferencia o deducción de la lógica como reglas de reescritura que transforman ciertos patrones de fórmulas en otros patrones, módulo los axiomas estructurales dados.

Muchos lenguajes y sistemas pueden especificarse en lógica de reescritura de forma natural utilizando técnicas similares, construyendo así prototipos de ejecución de dichos lenguajes. En particular, las similitudes entre la lógica de reescritura y las semánticas operacionales estructurales [Plo81] se mostraron por primera vez en [Mes92], y se exploraron con más detalle en [MOM93]. Como ejemplo ilustrativo, en el artículo [MOM93] se desarrolló completamente una representación del lenguaje CCS de Milner [Mil89] utilizando lógica de reescritura, extendiendo las ideas presentadas anteriormente en [MFW92], y que estudiaremos en esta tesis en gran profundidad en los Capítulos 4 y 5. En la introducción

ya citamos trabajos relacionados con la utilización de la lógica de reescritura como marco semántico.

En este capítulo mostraremos las ideas generales de utilización de la lógica de reescritura y Maude presentadas en [MOM93] para representar semánticas operacionales estructurales. Veremos los problemas de ejecutabilidad que se presentan e ideas sobre soluciones que nosotros proponemos, que serán detalladas en los capítulos siguientes. También presentaremos las buenas propiedades de Maude como *metalenguaje* en el cual construir herramientas completas para otros lenguajes.

3.1. Semánticas operacionales estructurales

Las *semánticas operacionales estructurales* son un marco introducido originalmente por G. D. Plotkin [Plo81], en el cual la semántica operacional de un lenguaje de programación se especifica de forma lógica, independiente de la arquitectura de una máquina o de detalles de implementación, por medio de reglas que proporcionan una definición inductiva basada en la estructura de las expresiones del lenguaje. Dirigimos al lector al libro de M. Hennessy "*The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*" [Hen90], donde se presenta una introducción clara al tema.

Dentro de la semántica operacional estructural existen dos enfoques principales:

- *Semántica de paso largo* (*big-step semantics*, en inglés), también denominada *semántica natural* por Kahn [Kah87] y *semántica de evaluación* por Hennessy [Hen90]. En este enfoque, el predicado inductivo principal describe el resultado total o el valor de la ejecución de un cómputo hasta su terminación. Por esta razón, este enfoque no es apropiado para lenguajes como CCS donde se pretende que la mayoría de los programas no terminen.

- *Semántica de paso corto* (*small-step semantics*, en inglés), denominada también *semántica de computación* por Hennessy [Hen90]. En este enfoque, el predicado inductivo principal describe la ejecución de pasos individuales de un cómputo, siendo el cómputo total, en su caso, el cierre transitivo de tales pasos cortos. La semántica operacional de CCS presentada al comienzo del Capítulo 4 es un ejemplo.

Para ilustrar estas ideas, y como ejemplo sencillo que utilizaremos en las siguientes secciones, vamos a presentar la semántica de paso largo (o evaluación) de un lenguaje de expresiones aritméticas y booleanas, con distinción de casos y declaración de variables locales, denominado *Exp4* en [Hen90]. Su sintaxis abstracta, con significado intuitivo obvio, se describe en la Figura 3.1. Este lenguaje se extenderá a un lenguaje funcional simple en el Capítulo 6.

Los juicios de esta semántica serán de la forma

$$\rho \vdash e \Longrightarrow_A v$$

1. Categorías sintácticas

$$\begin{array}{rcl@{\qquad}rcl} op & \in & Op & bop & \in & BOp \\ n & \in & Num & & & \\ e & \in & Exp & be & \in & BExp \\ x & \in & Var & bx & \in & BVar \end{array}$$

2. Definiciones

$$\begin{array}{rcl} op & ::= & + \mid - \mid * \\ bop & ::= & And \mid Or \\ e & ::= & n \mid x \mid e'\ op\ e'' \mid \mathsf{If}\ be\ \mathsf{Then}\ e'\ \mathsf{Else}\ e'' \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e'' \\ be & ::= & bx \mid \mathsf{T} \mid \mathsf{F} \mid be'\ bop\ be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e, e') \end{array}$$

Figura 3.1: Sintaxis abstracta de *Exp4*.

donde ρ es un entorno que almacena el valor de cada variable, e es una expresión aritmética del lenguaje y v es el valor al que la expresión e se evalúa; o de la forma

$$\rho \vdash be \Longrightarrow_B bv$$

donde be es una expresión booleana y bv el valor al que se evalúa.

Las reglas semánticas de este lenguaje se muestran en las Figuras 3.2 y 3.3. Para una explicación de las mismas referimos al lector a [Hen90].

No obstante, sí comentaremos algún aspecto que va a hacer que modifiquemos la forma de algunas reglas en la representación en Maude de la Sección 3.2. En este tipo de semánticas es habitual que aparezcan reglas como nuestra OpR. Esta regla expresa que para evaluar la expresión $e\ op\ e'$ en el entorno ρ, hay que evaluar tanto e como e', para obtener sus valores v y v', respectivamente, siendo el resultado total la aplicación (con la función Ap) del operador binario op a los resultados v y v'. Si queremos utilizar la regla para probar si un juicio $\rho \vdash e\ op\ e' \Longrightarrow_A v''$ es válido, tendríamos que terminar comparando v'' con $Ap(op, v, v')$. Esto es lo que dice directamente la siguiente versión de la regla que incluye una condición lateral:

$$\text{OpR}' \quad \frac{\rho \vdash e \Longrightarrow_A v \qquad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash e\ op\ e' \Longrightarrow_A v''} \quad v'' = Ap(op, v, v')$$

La regla VarR también utiliza una expresión no trivial en la parte derecha del juicio. Podemos evitarlo si modificamos la regla de la siguiente manera:

$$\text{VarR}' \quad \frac{}{\rho \vdash x \Longrightarrow_A v} \quad v = \rho(x)$$

Lo mismo ocurre con las reglas BVarR y BOpR, cuyas modificaciones triviales no mostramos.

3.2. Reglas de inferencia como reescrituras

Al proponer la lógica de reescritura como marco lógico y semántico, Narciso Martí Oliet y José Meseguer mostraron en [MOM93] diversas formas diferentes de representar en ella

$$\text{CR}\quad \frac{}{\rho \vdash n \Longrightarrow_A n}$$

$$\text{VarR}\quad \frac{}{\rho \vdash x \Longrightarrow_A \rho(x)}$$

$$\text{OpR}\quad \frac{\rho \vdash e \Longrightarrow_A v \qquad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash e\ op\ e' \Longrightarrow_A Ap(op, v, v')}$$

$$\text{IfR}\quad \frac{\rho \vdash be \Longrightarrow_B \mathsf{T} \qquad \rho \vdash e \Longrightarrow_A v}{\rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \Longrightarrow_A v}$$

$$\frac{\rho \vdash be \Longrightarrow_B \mathsf{F} \qquad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \Longrightarrow_A v'}$$

$$\text{LocR}\quad \frac{\rho \vdash e \Longrightarrow_A v \qquad \rho[v/x] \vdash e' \Longrightarrow_A v'}{\rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \Longrightarrow_A v'}$$

Figura 3.2: Semántica de evaluación para *Exp4*, $\Longrightarrow_A$.

$$\text{BCR}\quad \frac{}{\rho \vdash \mathsf{T} \Longrightarrow_B \mathsf{T}} \qquad \frac{}{\rho \vdash \mathsf{F} \Longrightarrow_B \mathsf{F}}$$

$$\text{BVarR}\quad \frac{}{\rho \vdash bx \Longrightarrow_A \rho(bx)}$$

$$\text{BOpR}\quad \frac{\rho \vdash be \Longrightarrow_B bv \qquad \rho \vdash be' \Longrightarrow_B bv'}{\rho \vdash be\ bop\ be' \Longrightarrow_B Ap(bop, bv, bv')}$$

$$\text{EqR}\quad \frac{\rho \vdash e \Longrightarrow_A v \qquad \rho \vdash e' \Longrightarrow_A v}{\rho \vdash \mathsf{Equal}(e, e') \Longrightarrow_B \mathsf{T}} \qquad \frac{\rho \vdash e \Longrightarrow_A v \qquad \rho \vdash e' \Longrightarrow_A v'}{\rho \vdash \mathsf{Equal}(e, e') \Longrightarrow_B \mathsf{F}}\ v \neq v'$$

$$\text{NotR}\quad \frac{\rho \vdash be \Longrightarrow_B \mathsf{T}}{\rho \vdash \mathsf{Not}\ be \Longrightarrow_B \mathsf{F}} \qquad \frac{\rho \vdash be \Longrightarrow_B \mathsf{F}}{\rho \vdash \mathsf{Not}\ be \Longrightarrow_B \mathsf{T}}$$

Figura 3.3: Semántica de evaluación para expresiones booleanas, $\Longrightarrow_B$.

sistemas de inferencia. Una posibilidad muy general consiste en representar una regla de inferencia de la forma

$$\frac{S_1 \dots S_n}{S_0}$$

como una regla de reescritura de la forma $S_1 \dots S_n \longrightarrow S_0$, que reescribe *multiconjuntos* de juicios S_i. Esta transformación es correcta desde un punto de vista abstracto, pero al pensar en términos de ejecutabilidad de las reglas de reescritura, resulta más apropiado considerar reglas de reescritura invertidas de la forma $S_0 \longrightarrow S_1 \dots S_n$, que siguen reescribiendo multiconjuntos de juicios pero que van de la conclusión a las premisas, de forma que la reescritura con ellas corresponde a una búsqueda de una demostración de abajo arriba. De nuevo esta transformación es correcta, y en ambos casos, la idea intuitiva es que la relación de reescritura equivale a la línea horizontal que separa la conclusión de las premisas en la presentación habitual de las reglas de inferencia en los libros de texto.

En [MOM93] se sigue la primera idea (las premisas se reescriben a la conclusión) para representar en lógica de reescritura la semántica de CCS y del lenguaje funcional Mini-ML tomado del artículo de Kahn [Kah87] (véase Capítulo 6).

Nosotros aquí vamos a seguir la segunda idea, donde la conclusión se reescribe al conjunto de premisas. Vamos a concretar las representaciones utilizando el lenguaje Maude, aunque como veremos más adelante (Sección 3.4) los módulos resultantes no son directamente ejecutables en el sistema Maude.

Antes de ver la representación de la semántica operacional hemos de representar la sintaxis del lenguaje. Hay que apuntar que la especificación de la sintaxis del lenguaje está fuera del enfoque de la semántica operacional estructural. Sin embargo, gracias a la estructura de tipos ordenados de la lógica de reescritura, podemos dar tal especificación como el siguiente módulo funcional en Maude:

```
fmod SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp .

  op V : Qid -> Var .
  subsort Var < Exp .
  subsort Num < Exp .

  ops + - * : -> Op .

  op 0 : -> Num .
  op s : Num -> Num .

  op ___ : Exp Op Exp -> Exp [prec 20] .
  op let_=_in_ : Var Exp Exp -> Exp [prec 25] .
  op If_Then_Else_ : BExp Exp Exp -> Exp [prec 25] .

  op BV : Qid -> BVar .
  subsort BVar < BExp .
  subsort Boolean < BExp .
```

```
  ops T F : -> Boolean .
  ops And Or : -> BOp .
  op ___ : BExp BOp BExp -> BExp [prec 20] .
  op Not_ : BExp -> BExp [prec 15] .
  op Equal : Exp Exp -> BExp .
endfm
```

Utilizamos identificadores con comilla, de tipo `Qid`, para representar los identificadores de variables del lenguaje *Exp4*. En vez de declarar `Qid` como subtipo de `Var`, ya que este tipo también se utiliza para representar las variables booleanas, optamos por tener constructores `V` y `BV` que conviertan los `Qid` en valores de los tipos `Var` y `BVar`, respectivamente. Como constantes numéricas utilizamos los números naturales en notación de Peano, con constructores `0` y `s`.

También podemos definir en un módulo funcional `AP` la función de aplicación Ap de un operador binario a sus dos argumentos ya evaluados, utilizada en la definición de la semántica, y en otro módulo funcional `ENV` los entornos para conocer el valor de una variable (numérica o booleana). Estos dos módulos, al igual que el módulo `SYNTAX` que define la sintaxis del lenguaje, son independientes de la representación que hagamos de la semántica. Por tanto, los utilizaremos tanto en esta sección como en la siguiente, donde no los repetiremos.

```
fmod AP is
  protecting SYNTAX .

  op Ap : Op Num Num -> Num .

  vars n n' : Num .

  eq Ap(+, 0, n) = n .
  eq Ap(+, s(n), n') = s(Ap(+, n, n')) .
  eq Ap(*, 0, n) = 0 .
  eq Ap(*, s(n), n') = Ap(+, n', Ap(*, n, n')) .
  eq Ap(-, 0, n) = 0 .
  eq Ap(-, s(n), 0) = s(n) .
  eq Ap(-, s(n), s(n')) = Ap(-, n, n') .

  op Ap : BOp Boolean Boolean -> Boolean .

  var bv bv' : Boolean .

  eq Ap(And, T, bv) = bv .
  eq Ap(And, F, bv) = F .
  eq Ap(Or, T, bv) = T .
  eq Ap(Or, F, bv) = bv .
endfm
```

```
fmod ENV is
  protecting SYNTAX .

  sorts Value Variable .

  subsorts Num Boolean < Value .
  subsorts Var BVar < Variable .

  sort ENV .

  op mt : -> ENV .
  op _=_ : Variable Value -> ENV [prec 20] .
  op __ : ENV ENV -> ENV [assoc id: mt prec 30] .
  op _`(_`) : ENV Variable -> Value .
  op _`[_/_`] : ENV Value Variable -> ENV [prec 35] .
  op remove : ENV Variable -> ENV .

  vars X X' : Variable .
  var V : Value .
  var ro : ENV .

  eq (X = V ro)(X') = if X == X' then V else ro(X') fi .
  eq ro [V / X] = remove(ro, X) X = V .
  eq remove(mt, X) = mt .
  eq remove(X = V ro, X') = if X == X' then ro else X = V remove(ro,X') fi .

endfm
```

Las operaciones `mt`, `_=_` y `__` (del módulo `ENV`) se utilizan para construir entornos vacíos, unitarios y uniones de entornos, respectivamente. La operación `_(_)` se utiliza para consultar el valor asociado a una variable en un entorno, y se define de forma recursiva mediante una ecuación. La operación `_[_/_]` se utiliza para modificar la asociación entre una variable y un valor en un entorno, y se define mediante la operación auxiliar `remove` que elimina de un entorno una variable dada.

Ahora podemos representar la semántica operacional. Un juicio $\rho \vdash e \Longrightarrow_A v$ se representa mediante un término `ro |- e ==>A v` de tipo `Judgement`, construido mediante el siguiente operador

```
  op _|-_==>A_ : ENV Exp Num -> Judgement [prec 50] .
```

De igual forma tendremos un operador para construir juicios de la forma $\rho \vdash be \Longrightarrow_B bv$.

```
  op _|-_==>B_ : ENV BExp Boolean -> Judgement [prec 50] .
```

En general, una regla semántica tiene una conclusión y un conjunto de premisas, cada una de las cuales se representa mediante un juicio o sentencia. Por tanto se necesita un tipo de datos para representar conjuntos de juicios:

```
  sort JudgementSet .
  subsort Judgement < JudgementSet .

  op emptyJS : -> JudgementSet .
  op __ : JudgementSet JudgementSet -> JudgementSet
          [assoc comm id: emptyJS prec 60] .
```

El constructor de unión se escribe con sintaxis vacía (`__`), y se declara como asociativo (`assoc`), conmutativo (`comm`), y teniendo como elemento neutro el conjunto vacío (`id: emptyJS`). El encaje de patrones y la reescritura tendrán lugar *módulo* dichas propiedades. La propiedad de idempotencia se especifica mediante una ecuación explícita, ya que Maude no soporta la combinación del atributo de idempotencia con los atributos de asociatividad y conmutatividad.

```
  var J : Judgement .
  eq J J = J .
```

Una regla semántica se implementa como una regla de reescritura en la que el conjunto unitario formado por el juicio que representa la conclusión se reescribe al conjunto formado por los juicios que representan las premisas. Los esquemas de axioma (reglas semánticas sin premisas), como CR y BCR, se representan por medio de reglas de reescritura que reescriben la representación de la conclusión al conjunto vacío de juicios. Si la regla semántica tiene una condición lateral, esta se representa como la condición de una regla de reescritura condicional.

El siguiente módulo de sistema `EVALUATION` incluye la representación de las semánticas operacionales estructurales para expresiones aritméticas y booleanas de las Figuras 3.2 y 3.3, con las modificaciones de las reglas comentadas al final de la sección anterior. Utilizando el hecho de que cualquier texto que empiece por `---` es un comentario en Maude, la regla se presenta de una forma que resalta la correspondencia con la presentación usual en los libros de texto (aunque en este caso la conclusión se encuentra debajo de la línea horizontal). Esta notación se debe a K. Futatsugi. El texto que comienza por `***` también es un comentario.

```
mod EVALUATION is
  protecting AP .
  protecting ENV .

  var n : Num .
  var x : Var .
  vars e e' : Exp .
  var op : Op .
  var v v' v'' : Num .
  var ro : ENV .
  var bv bv' bv'' : Boolean .
  var bx : BVar .
  vars be be' : BExp .
  var bop : BOp .
```

```
sort Judgement .
op _|-_==>A_ : ENV Exp Num -> Judgement [prec 50] .
op _|-_==>B_ : ENV BExp Boolean -> Judgement [prec 50] .

sort JudgementSet .
subsort Judgement < JudgementSet .
op emptyJS : -> JudgementSet .
op __ : JudgementSet JudgementSet -> JudgementSet
        [assoc comm id: emptyJS prec 60] .


*** Evaluation semantics for Exp4

rl [CR] :   ro |- n ==>A n
       => ------------------
             emptyJS .

crl [VarR'] :   ro |- x ==>A v
           => ------------------
                   emptyJS
           if v == ro(x) .

crl [OpR'] :   ro |- e op e' ==>A v''
          => -------------------------
               ro |- e ==>A v
               ro |- e' ==>A v'
          if v'' == Ap(op, v, v') .

rl [IfR1] :  ro |- If be Then e Else e' ==>A v
         => ----------------------------------
              ro |- be ==>B T
              ro |- e  ==>A v .

rl [IfR2] :  ro |- If be Then e Else e' ==>A v'
         => ----------------------------------
              ro |- be ==>B F
              ro |- e'  ==>A v' .

rl [LocR] :  ro |- let x = e in e' ==>A v'
         => -------------------------------
              ro |- e  ==>A v
              ro[v / x] |- e' ==>A v' .


*** Evaluation semantics for boolean expressions

rl [BCR1] :  ro |- T ==>B T
         => ----------------
               emptyJS .
```

```
  rl [BCR2] :  ro |- F ==>B F
            => ----------------
                   emptyJS .

  crl [BVarR'] :   ro |- bx ==>B bv
               => -------------------
                        emptyJS
               if bv == ro(bx) .

  crl [BOpR'] :   ro |- be op be' ==>B bv''
              => ------------------------
                   ro |- be ==>B bv
                   ro |- be' ==>B bv'
              if bv'' == Ap(bop, bv, bv') .

  rl [EqR1] :  ro |- Equal(e,e') ==>B T
            => -------------------------
                 ro |- e  ==>A v
                 ro |- e' ==>A v .

  crl [EqR2] :  ro |- Equal(e,e') ==>B F
             => --------------------------
                  ro |- e  ==>A v
                  ro |- e' ==>A v'
             if v =/= v' .

  rl [NotR1] :  ro |- Not be ==>B F
             => ---------------------
                  ro |- be ==>B T .

  rl [NotR2] :  ro |- Not be ==>B T
             => ---------------------
                  ro |- be ==>B F .
endm
```

Esta representación de la semántica operacional tiene un marcado carácter de *demostrador*, en el que el proceso de reescritura corresponde a encontrar una demostración de la corrección del juicio inicial que se reescribe.

Así, un juicio $\rho \vdash e \Longrightarrow_A v$ es *válido*, o sea derivable a partir de las reglas semánticas para *Exp4*, si y solo si el término de tipo `Judgement` que lo representa puede ser reescrito al conjunto vacío de juicios utilizando las reglas de reescritura del módulo `EVALUATION` que definen la semántica operacional. Intuitivamente, la idea es que se empieza con un juicio que quiere probarse que es válido y se trabaja hacia atrás utilizando el proceso de reescritura de una forma dirigida por objetivos, manteniendo el conjunto de juicios que deben cumplirse y faltan por demostrar para probar la corrección del juicio inicial. De modo que si este conjunto llega a ser vacío habremos concluido que el juicio inicial era correcto.

La representación conseguida mediante este enfoque, aunque correcta desde un punto de vista teórico, presenta diversos problemas a la hora de ser ejecutada, como veremos en la Sección 3.4.

3.3. Transiciones como reescrituras

La transformación presentada en la sección anterior se puede aplicar a una gran variedad de sistemas de inferencia, como se muestra detalladamente en [MOM93], incluyendo sistemas de secuentes para lógicas y también definiciones de semánticas operacionales para lenguajes. Sin embargo, en el caso de las semánticas operacionales, los juicios S_i típicamente tienen la forma de algún tipo de "transición" $P \rightarrow Q$ entre estados, de forma que tiene sentido considerar la posibilidad de transformar esta relación de transición entre estados en la relación de reescritura entre términos que representen los estados. Esta posibilidad también se contempló en [MOM93]. Cuando pensamos de esta manera, una regla de inferencia de la forma

$$\frac{P_1 \rightarrow Q_1 \ldots P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$$

se convierte en una regla de reescritura *condicional* de la forma

$$P_0 \longrightarrow Q_0 \quad \textit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n,$$

donde la condición incluye las reescrituras correspondientes a las premisas de la regla semántica.

Las reglas de esta forma ya fueron consideradas por José Meseguer en su artículo sobre lógica de reescritura [Mes92] y, como se vio en la Sección 2.4, pueden utilizarse directamente en Maude 2.0.

En el caso concreto de la semántica operacional del lenguaje *Exp4*, los tipos de los términos a la izquierda y a la derecha de la flecha que representa la transición no son iguales, debido a que a la izquierda aparece también el entorno ρ. Utilizaremos un tipo `Statement` para representar el par formado por un entorno y una expresión.

```
sort Statement .
op _|-_ : ENV Exp -> Statement [prec 40] .
```

A la derecha de la transición aparece siempre un valor (numérico o booleano). Como en este tipo de representación se reescribirán términos de tipo `Statement` en términos de tipo `Num` o `Boolean`, estos tipos tienen que pertenecer a la misma componente conexa.

```
subsorts Num Boolean < Statement .
```

El siguiente módulo de sistema `EVALUATION2` contiene la representación de la semántica operacional del lenguaje *Exp4* siguiendo el enfoque en el que las transiciones semánticas se representan mediante reescrituras.

```
mod EVALUATION2 is
  protecting AP .
  protecting ENV .

  sort Statement .
  subsorts Num Boolean < Statement .
  op _|-_ : ENV Exp -> Statement [prec 40] .
  op _|-_ : ENV BExp -> Statement [prec 40] .

  var ro : ENV .
  var n : Num .
  var x : Var .
  var bx : BVar .
  var v v' : Num .
  var bv bv' : Boolean .
  var op : Op .
  vars e e' : Exp .
  vars be be' : BExp .
  var bop : BOp .

  *** Evaluation semantics for Exp4

  rl [CR] : ro |- n => n .

  rl [VarR] : ro |- x => ro(x) .

  crl [OpR] : ro |- e op e' => Ap(op,v,v')
           if ro |- e  => v  /\
              ro |- e' => v' .

  crl [IfR1] : ro |- If be Then e Else e' => v
            if ro |- be => T  /\
               ro |- e  => v .

  crl [IfR2] : ro |- If be Then e Else e' => v'
            if ro |- be => F  /\
               ro |- e' => v' .

  crl [LocR] : ro |- let x = e in e' => v'
            if ro |- e  => v /\
               ro[v / x] |- e' => v' .

  *** Evaluation semantics for boolean expressions

  rl [BCR1] : ro |- T => T .

  rl [BCR2] : ro |- F => F .

  rl [BVarR] : ro |- bx => ro(bx) .
```

```
  crl [BOpR] : ro |- be bop be' => Ap(bop,bv,bv')
           if ro |- be  => bv  /\
              ro |- be' => bv' .

  crl [EqR1] : ro |- Equal(e,e') => T
           if ro |- e  => v  /\
              ro |- e' => v .

  crl [EqR2] : ro |- Equal(e,e') => F
           if ro |- e  => v  /\
              ro |- e' => v' /\ v =/= v' .

  crl [NotR1] : ro |- Not be => F
            if ro |- be => T .

  crl [NotR2] : ro |- Not be => T
            if ro |- be => F .
endm
```

En esta representación no hemos tenido que utilizar las modificaciones de las reglas semánticas VarR, OpR, BVarR y BOpR, ya que el funcionamiento de las reglas de reescritura correspondientes es el mismo que la lectura que hacíamos de las reglas semánticas. Por ejemplo, la regla `OpR` dice que una expresión `e op e'` se puede reescribir si sus subexpresiones se pueden reescribir a valores `v` y `v'` y, si esto es así, el valor al que se reescribe la expresión original se calcula aplicando el operador `op` a dichos valores.

Ahora la idea del proceso de reescritura es diferente. Un juicio de la semántica operacional $\rho \vdash e \Longrightarrow_A v$ es *válido* si y solo si el término que representa la parte de la izquierda `ro |- e` puede ser reescrito al término `v` utilizando las reglas de reescritura del módulo `EVALUATION2`. En cierto sentido, podemos ver esta representación de la semántica más como un *intérprete* del lenguaje *Exp4*, donde dado un entorno y una expresión, estos se reescriben obteniéndose el valor al que se reduce la expresión según la semántica operacional de evaluación del lenguaje. Por ejemplo, podemos pedir a Maude que reescriba la expresión $\mathsf{s}(\mathsf{0}) + x$ en un entorno en el que x vale $\mathsf{s}(\mathsf{0})$:

```
Maude> rew (V('x) = s(0)) |- s(0) + V('x) .
result Num: s(s(0))
```

Esta visión de la representación semántica como intérprete es diferente de la visión de *demostrador* de la corrección de un juicio que tenía la representación de la sección anterior. Sin embargo, utilizando el comando `search` de Maude 2.0, que busca entre las posibles reescrituras de un término aquellas que encajen con un patrón dado, podemos obtener este carácter de demostrador. Por ejemplo, con la siguiente ejecución,

```
Maude> search (V('x) = s(0)) |- s(0) + V('x) => s(s(0)) .

Solution 1 (state 1)
empty substitution
```

```
No more solutions.
```

al haber obtenido una solución, queda probado que el juicio inicial es correcto.

Desde un punto de vista teórico, esta forma de representar semánticas es siempre lógicamente correcta. Desde un punto de vista práctico, es directamente utilizable para algunas semánticas, como, por ejemplo, para la semántica del lenguaje *Exp4* como demuestran los ejemplos anteriores. Sin embargo, como veremos en la Sección 3.4.2, esta forma de representación también puede tener problemas de ejecutabilidad.

3.4. Maude como marco semántico ejecutable

Uno de los objetivos que queremos conseguir (el más importante de esta tesis) al representar en Maude la semántica operacional de un lenguaje es que la representación obtenida sea *ejecutable* y podamos utilizarla tanto desde un punto de vista más computacional, como *intérprete* que nos permita tener un prototipo del lenguaje, como desde un punto de vista más lógico, como *demostrador* que nos permita obtener pruebas de la corrección de un juicio, a partir de su derivabilidad utilizando las reglas semánticas.

Como hemos dicho anteriormente, los dos tipos generales de representación mostrados en las secciones anteriores adolecen de una serie de problemas de ejecutabilidad, en las versiones actuales de Maude. Al comenzar el trabajo de esta tesis nos propusimos el objetivo de conseguir, a partir del conjunto de reglas de inferencia que definen la semántica operacional estructural de un lenguaje, una representación en Maude completamente ejecutable que resolviera estos problemas. El ejemplo sobre el que estudiamos los problemas y presentamos soluciones fue la semántica operacional de CCS [Mil89] y la lógica modal de Hennessy-Milner [HM85, Sti96] para describir propiedades de procesos CCS. Este caso de estudio se presentará con todo detalle en los Capítulos 4 y 5. Los problemas encontrados y las soluciones propuestas no son exclusivos de CCS (de hecho se aplican de igual forma para representar la semántica de la lógica modal mencionada), y pueden aparecer en muchas de las representaciones de semánticas de lenguajes que hagamos siguiendo los dos enfoques vistos.

En esta sección mostraremos los problemas de ejecutabilidad que sufren ambos enfoques y presentaremos las ideas generales detrás de las soluciones propuestas, que serán discutidas con mayor profundidad para el caso de CCS en los Capítulos 4 y 5.

3.4.1. Reglas de inferencia como reescrituras

Primero discutiremos los problemas que aparecen al trabajar con el enfoque que convierte los juicios de la semántica en términos y las reglas de inferencia en reglas de reescritura.

El primer problema con el que nos encontramos en este enfoque es la presencia de variables *nuevas* en las premisas, es decir en la parte derecha de la regla de reescritura, que no aparecían en la conclusión, parte izquierda de la regla de reescritura. Por ejemplo, en la regla `LocR` para el operador de declaración local de variables

```
rl [LocR] :  ro |- let x = e in e' ==>A v'
          => -------------------------------
               ro |- e  ==>A v
               ro[v / x] |- e' ==>A v' .
```

la variable `v` aparece solo en la parte derecha. Este tipo de reglas no pueden ser utilizadas directamente por el intérprete por defecto de Maude. Además, aunque en la nueva versión de Maude 2.0 sí es posible manejar variables nuevas en algunos casos, esta regla *no* sería *admisible* (véase Sección 2.4 y [CDE+00b]).

Este problema puede resolverse utilizando el concepto de *metavariables explícitas* presentado en [SM99] en un contexto muy similar. Las variables nuevas en la parte derecha de una regla de reescritura representan valores "desconocidos" cuando se reescribe; utilizando metavariables se hace explícita esta falta de conocimiento. La semántica con metavariables explíticas se encarga de ligarlas a valores concretos una vez estos son conocidos.

Como el número de metavariables diferentes no se conoce de antemano, ya que no depende del número de reglas sino de la profundidad de las demostraciones y de las reglas que se utilicen para dichas demostraciones, necesitamos un mecanismo de generación de *metavariables nuevas*, que genere metavariables no utilizadas cada vez que se aplica una regla con metavariables. La reescritura que genera la demostración debe estar controlada por una *estrategia* al metanivel que genere nuevas metavariables (véase Sección 4.3.2).

En el Capítulo 4 veremos una representación ejecutable completa de la semántica de CCS utilizando metavariables.

La utilización de metavariables para resolver el problema de nuevas variables en la parte derecha de una regla, y la adaptación de la representación para tratar dichas metavariables, tiene como efecto lateral que podamos utilizar la representación obtenida como *intérprete* que calcule, dada una expresión, el valor al que esta se evalúa, sin tener que empezar con un juicio completo que incluya este valor. Si, por ejemplo, queremos evaluar la expresión `e` en un entorno de variables `ro`, podemos reescribir el juicio

```
ro |- e ==>A ?('result)
```

donde `?('result)` es una metavariable. En el proceso de reescritura de este juicio dicha metavariable se ligará al valor resultado de evaluar `e`. Añadiendo un mecanismo que almacene las ligaduras producidas en el proceso de demostración, podríamos, al terminar el proceso, examinar a qué valor se ha ligado la metavariable `?('result)`. Esto se hace en el Capítulo 4 para calcular todos los sucesores en un paso de un proceso CCS. Utilizando las ideas presentadas en la Sección 3.5 podríamos ocultar estos mecanismos, creando un intérprete para el lenguaje que evaluara expresiones.

Otro problema que presenta esta representación de las reglas de inferencia como reglas de reescritura es que algunas veces es posible aplicar más de una regla para reescribir un juicio y los términos a los que se llega tras la aplicación de estas reglas son diferentes. Es decir, el conjunto de reglas no es determinista.

Por ejemplo, para reducir el juicio

```
V('x) = 0 |- If Equal(V('x),0) Then s(0) Else V('x) ==>A s(0)
```

se pueden aplicar tanto la regla `IfR1` como `IfR2`.

En general, solo algunos de los caminos nos llevarán al conjunto vacío de juicios, en caso de que el juicio inicial sea probable. Por tanto, se hace necesario tratar con el árbol completo de posibles reescrituras de un juicio, buscando si una de las ramas conduce a `emptyJS`.

En la Sección 4.3.2 se describe con todo detalle cómo implementar, de forma suficientemente general, una estrategia de búsqueda en profundidad que recorre el árbol completo de posibles reescrituras de un término buscando nodos que cumplan cierto criterio de búsqueda. La estrategia presentada en dicha sección viene parametrizada por el módulo Maude que contiene las reglas de reescritura necesarias para construir el árbol de reescrituras (para pasar de un nodo a sus hijos) y por el predicado que indica cuándo un nodo representa un término que estamos buscando. En el caso concreto de aplicación de esta estrategia de búsqueda general para ver si un juicio es probable, el módulo será aquel con la representación de las reglas de inferencia y el predicado de búsqueda será cierto cuando el término represente el conjunto vacío de juicios, es decir, sea igual a `emptyJS`.

Los problemas encontrados con esta representación son intrínsecos al tipo de representación en sí, por lo que aparecerían en la representación de casi cualquier semántica operacional estructural no trivial que siga el enfoque de reglas de inferencia como reescrituras.

3.4.2. Transiciones como reescrituras

Este segundo enfoque está mucho más cercano a la idea de semántica operacional. Esto hace que los problemas de ejecutabilidad no aparezcan con tanta facilidad. De hecho, la representación dada en el módulo `EVALUATION2` de la Sección 3.3 es directamente ejecutable en Maude 2.0. En buena medida, los problemas que encontrábamos en el primer enfoque ya están resueltos por el propio sistema, al permitir el uso de reescrituras en las condiciones. La presencia de variables nuevas en la parte derecha de una regla de reescritura o en su condición está permitida y soportada (aunque con algunas restriciones, como vimos en la Sección 2.4), y la resolución de las reescrituras en las condiciones requiere búsqueda en el árbol de posibles reescrituras, también soportada por Maude 2.0.

Sin embargo, este tipo de representaciones también tiene algunos problemas de ejecutabilidad, que tienen que ver sobre todo con reescrituras no deseadas que no corresponden a transiciones, o búsquedas infinitas en la resolución de reescrituras en las condiciones sin éxito. En el Capítulo 5 veremos qué problemas hemos encontrado en la implementación de la semántica de CCS, y cómo estos problemas han podido ser resueltos de forma satisfactoria.

Uno de los problemas que podemos encontrar, que tiene que ver más con la representación en sí, y no con su ejecución, es que a menudo hay que ser cuidadoso para tener en cuenta cierta información adicional que aparece en las transiciones de la semántica operacional. Por ejemplo, en las semánticas operacionales estructurales para álgebras de procesos es esencial que las transiciones tengan alguna información de etiquetado que proporcione los mecanismos de sincronización. En el artículo [MOM93] ya se mostró cómo

resolver estos detalles para el caso particular de la semántica de CCS de Milner [Mil89], y nosotros lo veremos con detalle en el Capítulo 5.

Otro problema de este enfoque es que pueden generarse reescrituras que no corresponden a transiciones en la semántica operacional. El problema aparece si quisiéramos razonar sobre las posibles reescrituras de una expresión, o si una expresión pudiera dar lugar a un cómputo infinito o pudiera reescribirse a infinitos términos ninguno de los cuales fuera correcto. La solución está en utilizar el atributo `frozen` (véase Sección 2.4), que declara los argumentos del operador *congelados* y hace que el sistema no los reescriba.

En ocasiones se genera una pérdida de información al representar diferentes "flechas" o transiciones de la semántica operacional utilizando siempre la misma "flecha" o relación de reescritura. Este hecho ya ha ocurrido con la semántica del lenguaje *Exp4* cuando las transiciones $\Longrightarrow_A$ y $\Longrightarrow_B$ se han representado mediante reescrituras. Ya que los términos que pueden aparecer a la izquierda de dichas transiciones son disjuntos, no hemos tenido el problema de que las transiciones se mezclaran. Pero esta pérdida de información puede ser un problema, que podemos evitar si utilizamos un operador que distinga los términos a la izquierda de las diferentes transiciones.

3.5. Maude como metalenguaje

El uso de Maude como *metalenguaje* es una consecuencia natural de las buenas propiedades de la lógica de reescritura como marco semántico y lógico. Resumiremos en esta sección las ideas sobre Maude como metalenguaje presentadas en los artículos [CDE+98b, CDE+01].

Lo que tienen en común todas las aplicaciones de la lógica de reescritura como marco semántico y lógico es que los modelos de computación, las lógicas o los lenguajes se representan en lógica de reescritura por medio de funciones de la forma

$$(\dagger) \quad \Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

Estas representaciones son habitualmente muy simples y naturales, como hemos visto en las secciones anteriores y hacen corresponder teorías de reescritura a las teorías o módulos en $\mathcal{L}$.

Al ser reflexivo, un lenguaje basado en lógica de reescritura como Maude soporta funciones de representación de la forma (†), con lo que se convierte en un *metalenguaje* en el cual se puede *definir semánticamente* y *ejecutar* una gran variedad de lenguajes de programación, especificación y diseño, y sistemas de computación o lógicos.

Como vimos en el Capítulo 2, el diseño y la implementación del lenguaje Maude hacen un uso sistemático del hecho de que la lógica de reescritura sea reflexiva, y proporcionan un soporte eficiente de cómputos reflexivos por medio del módulo `META-LEVEL`.

Utilizando el módulo `META-LEVEL` se puede tanto hacer *ejecutable* la función de representación Φ, como ejecutar la teoría de reescritura resultante, que representa una teoría o módulo en $\mathcal{L}$, consiguiendo por tanto una implementación del lenguaje $\mathcal{L}$ en Maude. Concretamente, podemos representar al metanivel una función de representación Φ de

la forma (†) definiendo un tipo abstracto de datos $\texttt{Module}_{\mathcal{L}}$ que represente módulos en el lenguaje $\mathcal{L}$. Ya que en el módulo `META-LEVEL` también existe un tipo `Module` cuyos términos representan teorías de reescritura, podemos entonces *internalizar* la función de representación Φ como una función

$$\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}$$

definida ecuacionalmente. De hecho, gracias al resultado general de Bergstra y Tucker [BT80], cualquier función de representación Φ computable puede ser especificada de esta forma mediante un número finito de ecuaciones terminantes y *Church-Rosser*.

Al tener definida en Maude esta función de representación, podemos ejecutar la teoría de reescritura $\Phi(M)$ asociada a la teoría o módulo M en $\mathcal{L}$. Esto se ha hecho, por ejemplo, para módulos Maude estructurados y módulos orientados a objetos en Full Maude [Dur99]. También se han producido herramientas que extienden Maude y comprueban la propiedad de Church-Rosser [DM00b], coherencia [Dur00a] y terminación [Dur00b] para especificaciones Maude. Pero podría hacerse de igual forma para una gran variedad de lenguajes y lógicas. En el Capítulo 7 utilizaremos este método para implementar una herramienta para especificaciones Full LOTOS con tipos de datos definidos en ACT ONE, y en el Capítulo 9 para dar semántica al lenguaje RDF de descripción de recursos web propuesto por el W3C.

Al definir el tipo de datos $\texttt{Module}_{\mathcal{L}}$ en una extensión de `META-LEVEL` podemos definir la sintaxis de $\mathcal{L}$ dentro de Maude, incorporando otros tipos de datos auxiliares para comandos y otras construcciones. Esto puede hacerse de forma sencilla y natural utilizando el interfaz léxico mixfijo y los tipos de datos predefinidos de Maude `Token` (cualquier identificador) y `Bubble` (cualquier cadena de identificadores). La intuición que hay detrás de estos tipos es que corresponden a partes de un módulo del lenguaje que solo pueden analizarse sintácticamente una vez esté disponible la grámatica introducida por la signatura del propio módulo. La idea es que para lenguajes que permiten módulos con sintaxis definida por el usuario (como el propio Maude, o ACT ONE, véase el Capítulo 7) es natural ver su sintaxis como combinación de dos niveles diferentes: lo que podríamos llamar la sintaxis de *alto nivel* del lenguaje, y la sintaxis *definida por el usuario* introducida en cada módulo. Los tipos de datos `Token` y `Bubble` nos permiten reflejar estos dos niveles.

Para ilustrar estos conceptos, supongamos que queremos definir la sintaxis de Maude en Maude. Consideremos el siguiente módulo Maude:

```
fmod NAT3 is
  sort Nat3 .
  op 0 : -> Nat3 .
  op s_ : Nat3 -> Nat3 .
  eq s s s 0 = 0 .
endfm
```

Nótese que las cadenas de caracteres recuadradas no son parte de la sintaxis de alto nivel de Maude. Solo pueden ser analizadas sintácticamente utilizando la gramática asociada a la signatura del módulo `NAT3`, es decir, los operadores `0` y `s_` y el tipo `Nat3`.

A continuación mostramos parte del módulo `MAUDE` que define la sintaxis de Maude para módulos funcionales, y el comando de reducción asociado, reflejando esta dualidad de niveles sintácticos.

```
fmod MAUDE is
  sort PreModule PreCommand .
  subsort Decl < DeclList .

  op fmod_is_endfm : Token DeclList -> PreModule .
  op eq_=_. : Bubble Bubble -> Decl .
  op red in_:_. : Token Bubble -> PreCommand .
  ...
endfm
```

Obsérvese cómo se declaran explícitamente operadores que corresponden a la sintaxis de alto nivel de Maude, y se representan como términos de tipo `Bubble` aquellas partes de un módulo que solo pueden analizarse con la sintaxis definida por el usuario, como, por ejemplo, los lados izquierdo y derecho de una ecuación.

El módulo funcional `NAT3` puede analizarse sintácticamente como un término de tipo `PreModule` (incluyendo términos de tipo `Bubble`) del módulo `MAUDE`. Este término puede seguir siendo analizado por una función como la siguiente:

```
op parse-premodule : PreModule -> Module? .
```

que utiliza la función predefinida `meta-parse` de `META-LEVEL` para construir un término de tipo `Module`.

Dado un `PreModule` M, `parse-premodule` genera, primero, un `Module` M' con la misma signatura que M pero sin ecuaciones; después, `parse-premodule` intenta transformar el `PreModule` M en un `Module` M'', reduciendo cada `Bubble` b en M a un `Term` t, donde t es el resultado, en el caso de que haya habido éxito, de `meta-parse`(M', b). En caso de fallo, se devuelve un término de error del supertipo `Module?` de `Module`.

Para proporcionar un entorno de ejecución satisfactorio para un lenguaje $\mathcal{L}$ en Maude, también tenemos que dar soporte a la entrada/salida y a un estado persistente que permita interactuar con el intérprete de $\mathcal{L}$ que queremos definir. Es decir, deberemos ser capaces de introducir definiciones de módulos, comandos de ejecución y obtener los resultados de las ejecuciones. El módulo `LOOP-MODE` (véase Sección 2.2.6) hace esto posible, utilizando conceptos orientados a objetos. Como consecuencia, un entorno para $\mathcal{L}$ en Maude típicamente será implementado por un módulo que incluya como submódulos tanto `META-LEVEL` como `LOOP-MODE`. Este módulo contendrá una clase `System` con un canal de entrada de tipo `Input` (declarado como un supertipo de `PreModule` y `PreCommand`), un canal de salida de tipo `QidList`, y un atributo de tipo `Database`. Este tipo `Database` será especializado dependiendo de cada lenguaje $\mathcal{L}$ de interés. Por ejemplo, para Maude contendría el estado actual de la base de datos de módulos ya introducidos en el sistema.

```
  sort Input .
  subsorts PreCommand PreModule < Input .
  class System | db : Database, input : Input, output : QidList .
```

Para cada lenguaje particular se especifican el tipo `Database` y las reglas de reescritura que definan el comportamiento de la herramienta para los diferentes comandos del lenguaje. Por ejemplo, en el caso de Maude, tendríamos una regla que procesa un `PreModule` una vez introducido en el sistema

```
var DB : Database .
var PM: PreModule .
var I : Identifier .
var B : Bubble .

rl [premodule] :
      < system : System | db : DB, input : PM >
   => < system : System | db : processPreModule(PM, DB),
                          input : empty > .
```

La función `processPreModule` intenta analizar sintácticamente el `PreModule` utilizando la función `parse-premodule` anterior y, si tiene éxito, introduce el `Module` resultante en la base de datos.

También habrá que definir reglas para tratar comandos introducidos por el usuario. Por ejemplo, la siguiente regla trata el comando `red in _:_`:

```
rl [reduce] :
      < system : System | db : DB, input : red in I : B . >
   => < system : System | input : empty,
                  output : meta-pretty-print(getModule(I, DB),
                            meta-reduce(getModule(I, DB),
                             meta-parse(getModule(I, DB), B))) > .
```

donde `getModule` es una función que extrae de la base de datos el módulo cuyo nombre es el primer argumento.

Estas ideas se han utilizado para construir una herramienta donde ejecutar especificaciones Full LOTOS con tipos de datos descritos en el lenguaje ACT ONE. Veremos todos los detalles en el Capítulo 7.

Capítulo 4

Ejecución y verificación de CCS en Maude

En este capítulo se exploran las características de la lógica de reescritura, y en particular del lenguaje Maude, como un marco semántico y lógico donde representar tanto la semántica del álgebra de procesos CCS de Milner [Mil89] como una lógica modal para describir capacidades locales de procesos CCS [HM85, Sti96]. Para ello seguiremos el enfoque denominado *reglas de inferencia como reescrituras*.

Como vimos en la Sección 3.2, una posibilidad muy general a la hora de representar sistemas de inferencia en lógica de reescritura consiste en representar una regla de inferencia de la forma

$$\frac{S_1 \dots S_n}{S_0}$$

como una regla de reescritura de la forma $S_0 \longrightarrow S_1 \dots S_n$ que reescribe multiconjuntos de juicios pasando de la conclusión a las premisas, de tal forma que la reescritura con estas reglas corresponde a una búsqueda de una demostración de abajo arriba.

Aunque ya fue presentada anteriormente una representación de la semántica de CCS en lógica de reescritura en [MOM93] siguiendo el enfoque equivalente que reescribe las premisas a la conclusión, esta representación, si bien correcta desde un punto de vista teórico, no es ejecutable directamente. Es más, no se puede utilizar para responder preguntas tales como cuáles son los sucesores de un proceso después de realizar una acción, lo que se precisa para definir la semántica de la lógica modal de Hennessy-Milner.

Con la disponibilidad de la primera versión del sistema Maude en 1999 [CDE+99], nosotros emprendimos el proyecto de implementar cuidadosamente y de una forma completamente *ejecutable* la semántica operacional de CCS para ver si las ideas que teóricamente funcionan lo hacían también en la práctica. El primer problema con el que nos encontramos fue que la primera versión de Maude no permitía reglas condicionales con reescrituras en las condiciones, que estaban restringidas a condiciones booleanas. Esto no nos permitió considerar la forma de la representación en la cual se ven las transiciones como reescrituras (véase Sección 3.3), con lo que estábamos restringidos a la posibilidad de ver las reglas de inferencia como reescrituras (o transiciones como juicios). Incluso en este

caso, un número de problemas tenían que ser resueltos, a saber, la presencia de variables nuevas en los lados derechos de las reglas y el control del no determinismo en la aplicación de las reglas. En este capítulo se muestra cómo estos problemas han sido resueltos de una forma elegante utilizando las propiedades reflexivas de Maude. Las características del metanivel de Maude nos han permitido salvar una distancia importante entre la teoría y la práctica, obteniéndose una representación completamente ejecutable de la semántica operacional de CCS, que puede utilizarse para analizar procesos CCS mediante la lógica modal de Hennessy-Milner [HM85].

Nos gustaría apuntar que el principal objetivo de este capítulo (y de la tesis en sí) es describir con todo detalle cómo se pueden obtener representaciones *ejecutables* en Maude a partir de definiciones de la semántica operacional de lenguajes de programación o álgebras de procesos, o de sistemas de inferencia para lógicas en general. Estas representaciones ejecutables pueden verse como *prototipos* de intérpretes para los lenguajes o demostradores para las lógicas, haciendo posible la experimentación con los diferentes lenguajes o lógicas, y también con diferentes semánticas para el mismo lenguaje. En este momento no estamos interesados en utilizar Maude como un demostrador de teoremas para obtener modelos abstractos de las semánticas que nos permitieran hacer metarazonamientos (como por ejemplo sobre la teoría de bisimulación para CCS), en contraste con los objetivos de la mayoría de los trabajos relacionados que citamos más tarde en la Sección 4.8 y otros que citamos en el Capítulo 1.

Parte del trabajo presentado en este capítulo ha sido publicado en [VMO00a, VMO00b] y se presentó por primera vez en la conferencia *Formal Description Techniques for Distributed Systems and Communications Protocols, FORTE 2000*, dando lugar a la publicación [VMO00c]. Una versión actualizada, mejorada y extendida de este trabajo está pendiente de aceptación para su publicación en la revista *Formal Methods and System Design* [VMO02a].

4.1. CCS

Realizamos a continuación una introducción muy breve del *Calculus of Communicating Systems* de Milner, CCS [Mil89]. Se asume un conjunto A de *nombres*; los elementos del conjunto $\overline{A} = \{\overline{a} \mid a \in A\}$ se llaman *co-nombres*, y los elementos de la unión (disjunta) $\mathcal{L} = A \cup \overline{A}$ son *etiquetas* utilizadas para dar nombre a acciones ordinarias. La función $a \mapsto \overline{a}$ se extiende a $\mathcal{L}$ definiendo $\overline{\overline{a}} = a$. Hay una acción especial conocida como *acción silenciosa* denotada por τ, utilizada para representar la evolución interna de un sistema y, en particular, la sincronización de dos procesos por medio de las acciones a y $\overline{a}$. El conjunto de *acciones* es $\mathcal{L} \cup \{\tau\}$. El conjunto de procesos se define de forma intuitiva como sigue:

- 0 es un proceso inactivo que no hace nada.
- Si α es una acción y P es un proceso, $\alpha.P$ es el proceso que realiza la acción α y después se comporta como P.

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \qquad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L} \quad \alpha \notin L \cup \overline{L} \qquad \frac{P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'} \quad X =_{def} P$$

Figura 4.1: Reglas de la semántica operacional de CCS.

- Si P y Q son procesos, entonces $P + Q$ es el proceso que puede comportarse como P o como Q.
- Si P y Q son procesos, entonces $P|Q$ representa a P y Q ejecutándose de forma concurrente con posibles comunicaciones a través de la sincronización del par de acciones ordinarias a y $\overline{a}$.
- Si P es un proceso y $f : \mathcal{L} \to \mathcal{L}$ es una función de reetiquetado tal que $f(\overline{a}) = \overline{f(a)}$, $P[f]$ es el proceso que se comporta como P pero con las acciones ordinarias renombradas de acuerdo con la función f.
- Si P es un proceso y $L \subseteq \mathcal{L}$ es un conjunto de acciones ordinarias, $P\backslash L$ es el proceso que se comporta como P pero con las acciones en $L \cup \overline{L}$ prohibidas.
- Si P es un proceso, X es un identificador de proceso, y $X =_{def} P$ es una ecuación de definición donde P puede referirse recursivamente a X, entonces X es un proceso que se comporta como P.

Esta explicación intuitiva puede hacerse más precisa en términos de la semántica operacional estructural mostrada en la Figura 4.1, que define un sistema etiquetado de transiciones para los procesos CCS.

4.2. Representación de CCS

En esta sección mostraremos cómo representar la semántica de CCS en Maude, para ver con detalle cómo aparecen los dos problemas mencionados anteriormente. Se utiliza Full Maude [Dur99] para sacar partido de los mecanismos de parametrización que ofrece.

Para empezar, se muestra la representación de la sintaxis de CCS en dos módulos funcionales mediante la definición de tipos de datos algebraicos que definen las acciones y los procesos de CCS. Se importa el módulo predefinido `QID` para utilizar los identificadores con comilla de tipo `Qid` para representar las etiquetas de CCS y los identificadores de proceso.

```
(fmod ACTION is
   protecting QID .

   sorts Label Act .
   subsorts Qid < Label < Act .

   op tau : -> Act .    *** silent action
   op ~_ : Label -> Label .

   var L : Label .
   eq ~ ~ L = L .
 endfm)

(fmod PROCESS is
   protecting ACTION .

   sorts ProcessId Process .
   subsorts Qid < ProcessId < Process .

   op 0 : -> Process .                                  *** inaction
   op _._ : Act Process -> Process [prec 25] .          *** prefix
   op _+_ : Process Process -> Process [prec 35] .      *** summation
   op _|_ : Process Process -> Process [prec 30] .      *** composition
   op _`[_/_`] : Process Label Label -> Process [prec 20] .
                        *** relabelling: [b/a] relabels "a" to "b"
   op _\_ : Process Label -> Process [prec 20] .        *** restriction
 endfm)
```

En la representación se considera CCS al completo, incluyendo definiciones de procesos mutuamente recursivos, por medio de *contextos*. Un contexto está bien definido si un identificador de proceso está definido como mucho una vez. Utilizando un axioma de pertenencia condicional (`cmb`) se establece qué valores del tipo `AnyContext` son contextos bien construidos, es decir, son de tipo `Context`. La evaluación de `def(X,C)` devuelve el proceso asociado al identificador de proceso `X` si este está definido; en otro caso, devuelve el proceso erróneo `not-defined`.

```
(fmod CCS-CONTEXT is
   protecting PROCESS .

   sorts AnyProcess Context AnyContext .
   subsort Process < AnyProcess .
   subsort Context < AnyContext .

   op _=def_ : ProcessId Process -> Context [prec 40] .
   op nil : -> Context .
   op _&_ : AnyContext AnyContext -> AnyContext [assoc comm id: nil prec 42] .
   op _definedIn_ : ProcessId Context -> Bool .
   op def : ProcessId Context -> AnyProcess .
   op not-defined : -> AnyProcess .
```

```
  op context : -> Context .

  vars X X' : ProcessId .
  var P : Process .
  var C : Context .

  cmb (X =def P) & C : Context if not(X definedIn C) .

  eq X definedIn nil = false .
  eq X definedIn ((X' =def P) & C) = (X == X') or (X definedIn C) .

  eq def(X, nil) = not-defined .
  eq def(X, ((X =def P) & C)) = P .
  ceq def(X, ((X' =def P) & C)) = def(X, C) if X =/= X' .
endfm)
```

La constante `context` se utiliza para mantener las definiciones de los identificadores de proceso utilizados en cada especificación concreta en CCS. La constante se declara aquí porque será utilizada en la definición de la semántica (Sección 4.3.1), pero será definida en un ejemplo posterior (véase la Sección 4.3.3).

A continuación, empezamos a definir la semántica de CCS.

```
(mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .
```

Siguiendo las ideas presentadas en el Capítulo 3, una transición CCS $P \xrightarrow{a} P'$ se representa en Maude mediante el término P `--` a `->` P' de tipo `Judgement`, construido mediante el operador

```
  sort Judgement .
  op _--_->_ : Process Act Process -> Judgement [prec 50] .
```

Utilizamos un tipo de datos para representar conjuntos de juicios y una operación de unión asociativa, conmutativa, con el conjunto vacío como elemento identidad, e idempotente:

```
  sort JudgementSet .
  subsort Judgement < JudgementSet .

  op emptyJS : -> JudgementSet .
  op __ : JudgementSet JudgementSet -> JudgementSet
          [assoc comm id: emptyJS prec 60] .

  var J : Judgement .
  eq J J = J .
```

Una regla semántica se implementa como una regla de reescritura donde el conjunto unitario formado por el juicio que representa la conclusión se reescribe al conjunto formado por los juicios que representan las premisas. Por ejemplo, para el operador de restricción de CCS tenemos la siguiente regla semántica (adaptación de la regla de la Figura 4.1 al caso en el que solo se restringe una etiqueta)[1],

$$\frac{P \xrightarrow{\alpha} P'}{P \setminus l \xrightarrow{\alpha} P' \setminus l} \; \alpha \neq l \wedge \alpha \neq \bar{l}$$

la cual se traduce a la siguiente regla de reescritura:

```
var L : Label .
var A : Act .
vars P P' Q Q' : Process .

crl [res] :  P \ L -- A -> P' \ L
        => ------------------------
                  P -- A -> P'      if (A =/= L) and (A =/= ~ L) .
```

Como ejemplo alternativo, el esquema de axioma

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

que define el comportamiento del operador de prefijo de CCS, da lugar a la siguiente regla de reescritura

```
rl [pref] :   A . P -- A -> P
        => ---------------------
                  emptyJS .
```

Por tanto, una transición $P \xrightarrow{a} P'$ es posible en CCS si y solo si el juicio que la representa puede ser reescrito al conjunto vacío de juicios utilizando reglas de reescritura de la forma descrita más arriba, que describen la semántica operacional de CCS.

Sin embargo, como introdujimos en la Sección 3.4, encontramos dos problemas al trabajar con este enfoque en la versión del sistema Maude 1.0.5. El primer problema es que en ocasiones aparecen variables nuevas en las premisas, que no aparecían en la conclusión. Por ejemplo, en la regla semántica para el operador paralelo correspondiente a la sincronización tenemos

```
rl [par] :     P | Q -- tau -> P' | Q'
        => -------------------------------
            P -- L -> P'   Q -- ~ L -> Q' .
```

[1]Podríamos generalizar fácilmente el operador de restricción para permitir ocultar un conjunto de etiquetas $P \setminus \{l_1, \ldots, l_n\}$, pero esto haría la sintaxis más compleja, sin añadir nada nuevo a la representación de la semántica. Lo mismo ocurre con el operador de renombramiento.

donde `L` es una variable nueva en la parte derecha de la regla de reescritura. Las reglas de este tipo no pueden ser utilizadas directamente por el intérprete por defecto de Maude; solo pueden utilizarse al metanivel utilizando una estrategia que instancie las variables nuevas, como veremos en la Sección 4.3.2.

Otro problema es que en ocasiones se pueden aplicar varias reglas para reescribir un mismo juicio. Por ejemplo, para el operador de suma, que es intrínsecamente no determinista, tenemos

```
rl [sum] :   P + Q -- A -> P'
         => --------------------
                P -- A -> P' .

rl [sum] :   P + Q -- A -> Q'
         => --------------------
                Q -- A -> Q' .
```

Si declarásemos el operador de CCS `_+_` como conmutativo, solo se necesitaría una regla. Sin embargo, el no determinismo seguiría apareciendo al haber varios posibles encajes (módulo conmutatividad) con el patrón `P + Q` en el término que representa una transición en la parte izquierda de la regla.

En general, no todas las posibles maneras de reescribir un juicio conducen al conjunto vacío de juicios. Por tanto, se hace necesario trabajar con el árbol completo de posibles reescrituras de un juicio, buscando si alguna de las ramas conduce a `emptyJS`.

4.3. Semántica de CCS ejecutable

En esta sección se muestra cómo resolver el problema de la presencia de variables nuevas en la parte derecha de una regla de reescritura utilizando el concepto de *metavariables explícitas* presentado en [SM99], y cómo controlar la reescritura no determinista por medio de una *estrategia de búsqueda* [BMM98, CDE+00c].

4.3.1. Definición de la semántica ejecutable

El problema de las variables nuevas en la parte derecha de una regla de reescritura se resuelve utilizando *metavariables* para representar valores *desconocidos*[2]. La semántica con metavariables explícitas se encarga de ligarlas a valores concretos una vez estos son conocidos.

Por ahora, solo es necesario el uso de metavariables para representar acciones en los juicios, por lo que declararemos un nuevo tipo de metavariables como acciones:

```
sort MetaVarAct .
```

[2]Nótese que este concepto de metavariables es distinto al utilizado en la Sección 2.2.5 al definir un lenguaje de estrategias. Allí las metavariables eran un concepto utilizado únicamente en el metanivel para asignar valor a variables del nivel objeto. Ahora las metavariables aparecen en el nivel objeto, aunque parte de su tratamiento (como la generación de nuevas metavariables) se hará al metanivel.

Para poder producir un número no determinado de metavariables vamos a utilizar el tipo `Qid` de los identificadores con comilla, y un operador constructor de metavariables:

```
op ?‘(_‘)A : Qid -> MetaVarAct .
var NEW1 : Qid .
```

Así, los términos cerrados `?('mv1)A`, `?('mv2)A`, `?('mv3)A`, ... representan metavariables diferentes. Para poder representar la idea de que nuevas metavariables deben ser generadas cada vez que se aplica una regla con variables nuevas a la derecha, en dichas reglas las metavariables aparecerán de la forma `?(NEW1)A`, como veremos más adelante.

También se introduce un nuevo tipo `Act?` de "posibles acciones", que engloba tanto las propias acciones como las metavariables como acciones, es decir, es supertipo de los tipos `Act` y `MetaVarAct`:

```
sort Act? .
subsorts Act MetaVarAct < Act? .
var ?A : MetaVarAct .
var A? : Act? .
```

Necesitamos modificar el operador de construcción de juicios para tratar con este nuevo tipo de acciones[3]:

```
op _--_->_ : Process Act? Process -> Judgement [prec 50] .
```

Como hemos dicho anteriormente, una metavariable será *ligada* cuando se conozca su valor concreto, por lo que es necesario un nuevo juicio que establezca que una metavariable está ligada a un valor concreto:

```
op ‘[_:=_‘] : MetaVarAct Act -> Judgement .
```

y un mecanismo de propagación de esta ligadura al resto de juicios donde la metavariable ligada pueda aparecer. Ya que esta propagación debe alcanzar a todos los juicios en el estado actual del proceso de inferencia, introducimos un operador para definir el conjunto completo de juicios, y una regla de reescritura para propagar la ligadura

```
sort Configuration .
op ‘{‘{_‘}‘} : JudgementSet -> Configuration .

var JS : JudgementSet .
rl [bind] : {{ [?A := A] JS }} => {{ <act ?A := A > JS }} .
```

donde se utilizan las siguientes funciones auxiliares sobrecargadas para realizar la correspondiente sustitución

[3]También podríamos haber sobrecargado el operador `_--_->_` mediante dos declaraciones, una en la que el segundo argumento fuera de tipo `Act` y otra en la que fuera de tipo `MetaVarAct`.

```
op <act_:=_>_ : MetaVarAct Act Act? -> Act? .
op <act_:=_>_ : MetaVarAct Act Judgement -> Judgement .
op <act_:=_>_ : MetaVarAct Act JudgementSet -> JudgementSet .
```

Estas funciones se definen ecuacionalmente de la siguiente manera:

```
eq <act ?A := A > ?A = A .
ceq <act ?A := A > ?A' = ?A' if ?A =/= ?A' .
eq <act ?A := A > A' = A' .
eq <act ?A := A > (~ A?) = ~(<act ?A := A > A?) .

eq <act ?A := A > (P -- A? -> Q) =  P -- <act ?A := A > A? -> Q .
eq <act ?A := A > [ ?A' := A' ] = [?A' :=  A'] .
eq <act ?A := A > [ A? =/= A'? ] =
   [<act ?A := A > A? =/= <act ?A := A > A'?] .

eq <act ?A := A > emptyJS = emptyJS .
ceq <act ?A := A > (J JS) = (<act ?A := A > J) (<act ?A := A > JS)
                          if JS =/= emptyJS .
```

Aunque en la regla `bind` los términos `[?A := A] JS` y `<act ?A := A > JS` pueden resultar muy parecidos, en realidad son muy diferentes. El primero representa un conjunto de juicios, uno de cuyos elementos (módulo conmutatividad y asociatividad) es el juicio `[?A := A]`. El segundo representa la aplicación de la operación modificadora de juicios `<act_:=_>_` al conjunto de juicios `JS`, y será reducido utilizando las ecuaciones anteriores.

Ahora ya podemos redefinir las reglas de reescritura que implementan la semántica de CCS, preocupándonos de las metavariables. Para el operador de prefijo mantenemos el axioma previo

```
rl [pref] :  A . P -- A -> P
        => --------------------
                  emptyJS .
```

y añadimos una nueva regla para el caso en el que una metavariable aparezca en el juicio

```
rl [pref] :  A . P -- ?A -> P
        => --------------------
                 [?A := A] .
```

Nótese cómo la metavariable `?A` presente en el juicio de la parte izquierda se liga a la acción concreta `A` tomada del proceso `A . P`. Esta ligadura se propagará a cualquier otro juicio dentro del conjunto de juicios que contuviera al juicio `A . P -- ?A -> P`.

Aunque hemos definido dos reglas para distinguir los casos en los que una acción o una metavariable aparece en la transición, la segunda regla sería suficiente si tuviéramos reglas auxiliares que convirtieran las transiciones en las que aparecen etiquetas que no son metavariables, introduciendo en su lugar una metavariable adicional que denote su valor. Comentaremos esta simplificación en la Sección 7.7.

Para el operador de suma se generaliza la regla permitiendo una variable `A?` más general de tipo `Act?`, ya que el comportamiento es el mismo, independientemente de si en el juicio aparece una metavariable o una acción:

```
rl [sum] :  P + Q -- A? -> P'
         => -------------------
               P -- A? -> P' .

rl [sum] :  P + Q -- A? -> Q'
         => -------------------
               Q -- A? -> Q' .
```

El no determinismo, que sigue estando presente, será tratado en la Sección 4.3.2.

Para el operador de composición paralela, tenemos dos reglas para los casos en los que uno de los procesos componentes realiza una acción por su cuenta,

```
rl [par] :  P | Q -- A? -> P' | Q
         => ------------------------
                 P -- A? -> P' .

rl [par] :  P | Q -- A? -> P | Q'
         => ------------------------
                 Q -- A? -> Q' .
```

y dos reglas adicionales más, para tratar el caso en el que se produce una comunicación entre ambos procesos:

```
rl [par] :            P | Q -- tau -> P' | Q'
         => ------------------------------------------
             P -- ?(NEW1)A -> P'   Q -- ~ ?(NEW1)A -> Q' .

rl [par] :              P | Q -- ?A -> P' | Q'
   => ---------------------------------------------------------
       P -- ?(NEW1)A -> P'   Q -- ~ ?(NEW1)A -> Q'  [?A := tau] .
```

donde también se ha sobrecargado el operador `~`

```
op ~_ : Act? -> Act? .
```

Nótese cómo se utiliza el término `?(NEW1)A` para representar una *metavariable nueva*. La reescritura debe estar controlada por una *estrategia* que instancie la variable `NEW1` con un nuevo identificador con comilla no utilizado anteriormente cada vez que se aplica una de las dos últimas reglas, con el fin de construir una metavariable *nueva*. La estrategia presentada en la Sección 4.3.2 logra esto. Obsérvese también que la variable `NEW1` es una variable *nueva* en la parte derecha de las reglas, es decir, continúa presente precisamente el problema que estamos intentando resolver. Lo que hemos logrado ha sido localizar la

presencia de estas variables, lo que permite definir mecanismos que instancien estas variables nuevas y propagen también sus valores cuando estos se conozcan. La idea principal es que cuando alguna de estas reglas sea aplicada realmente, el término `?(NEW1)A` habrá sido sustituido por un término *cerrado* como `?('mv1)A` que representará una metavariable nueva.

Hay dos reglas para tratar el operador de restricción de CCS: una para el caso en el que aparece una acción en el juicio de la parte izquierda y otra para el caso en el que aparece una metavariable en el juicio:

```
crl [res] :  P \ L -- A -> P' \ L
       => ------------------------
                 P -- A -> P'       if (A =/= L) and (A =/= ~ L) .

rl [res] :         P \ L -- ?A -> P' \ L
      => ----------------------------------------
          P -- ?A -> P'  [?A =/= L]  [?A =/= ~ L] .
```

En el primer caso, la condición lateral de la regla semántica se convierte en una condición de la regla de reescritura que la representa. Sin embargo, en el segundo caso no se puede utilizar una regla de reescritura condicional, porque la condición correspondiente `(?A =/= L) and (?A =/= ~ L)` no puede comprobarse hasta que se conozca el valor concreto de la metavariable `?A`. Por tanto, tenemos que añadir un nuevo tipo de juicios

```
op `[_=/=_`] : Act? Act? -> Judgement .
```

utilizado para establecer restricciones de desigualdad sobre los valores de las metavariables (las cuales serán sustituidas más tarde por acciones). La restricción se elimina cuando se cumple,

```
crl [dist] : [A =/= A'] => emptyJS  if A =/= A' .
```

donde se están utilizando acciones "normales", no metavariables, `A` y `A'`.

Para el operador de renombramiento de CCS se tienen reglas de reescritura similares, donde utilizamos las mismas técnicas.

```
rl [rel] :  P[M / L] -- M -> P'[M / L]
       => ----------------------------
                  P -- L -> P' .

rl [rel] :  P[M / L] -- ~ M -> P'[M / L]
       => -----------------------------
                 P -- ~ L -> P' .

crl [rel] :  P[M / L] -- A -> P'[M / L]
        => ----------------------------
                  P -- A -> P'     if (A =/= L) and (A =/= ~ L) .
```

```
rl [rel] :  P[M / L] -- ?A -> P'[M / L]
        => ----------------------------
             (P -- L -> P') [?A := M] .

rl [rel] :  P[M / L] -- ?A -> P'[M / L]
        => -----------------------------
            P -- ~ L -> P'  [?A := ~ M] .

rl [rel] :      P[M / L] -- ?A -> P'[M / L]
        => ----------------------------------------
             P -- ?A -> P' [?A =/= L] [?A =/= ~ L] .
```

Finalmente, para los identificadores de proceso, solo necesitamos generalizar la regla original por medio de una variable más general `A?`:

```
crl [def] :         X -- A? -> P'
       => -------------------------------
           def(X, context) -- A? -> P'   if X definedIn context .
```

Utilizando las reglas presentadas podemos empezar a proponer algunas preguntas sobre la capacidad de un proceso para realizar una acción. Por ejemplo, podemos preguntar si el proceso `'a.'b.0` puede realizar la acción `'a` (convirtiéndose en el proceso `'b.0`) reescribiendo la configuración formada por el juicio que representa dicha transición:

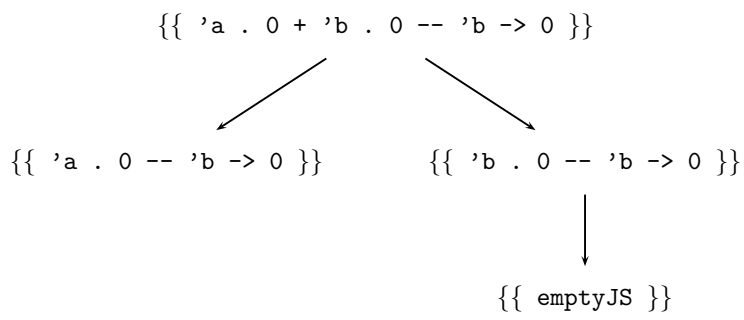
```
Maude> (rewrite {{ 'a . 'b . 0 -- 'a -> 'b . 0 }} .)
result Configuration : {{ emptyJS }}
```

Ya que se alcanza una configuración formada por el conjunto vacío de juicios, podemos concluir que la transición es válida.

Sin embargo, si preguntamos si el proceso `'a.0 + 'b.0` puede realizar la acción `'b` convirtiéndose en el proceso `0`, obtenemos el resultado

```
Maude> (rewrite {{ 'a . 0 + 'b . 0 -- 'b -> 0 }} .)
result Configuration : {{ 'a . 0 -- 'b -> 0 }}
```

el cual representa que la transición dada no es posible, lo cual no es cierto. El problema proviene del hecho de que la configuración `{{ 'a.0 + 'b.0 -- 'b -> 0 }}` se puede reescribir de dos formas diferentes, y solo una de ellas conduce a la configuración formada por el conjunto vacío de juicios, como muestra el siguiente árbol:



Por tanto, necesitamos una estrategia para buscar en el árbol de posibles reescrituras la configuración formada por el conjunto vacío de juicios.

4.3.2. Búsqueda en el árbol de reescrituras

En esta sección se muestra cómo pueden utilizarse las propiedades reflexivas de Maude [CM96] para controlar la reescritura de un término y la búsqueda en el árbol de posibles reescrituras de un término. La búsqueda en profundidad utilizada se basa en los trabajos presentados en [Bru99, BMM98, CDE+00c], aunque modificada para tratar la sustitución de metavariables explicada en la sección anterior. La estrategia que implementa la búsqueda es completamente general, por lo que su definición no viene determinada de ninguna forma por la representación presentada de CCS. La estrategia está parametrizada con respecto al módulo utilizado para construir el árbol de reescrituras, que incluye en particular el predicado que define los objetivos de la búsqueda. Nosotros utilizaremos como valor del parámetro el módulo `CCS-SEMANTICS` para instanciar la estrategia de búsqueda en la Sección 4.3.3 y el módulo `MODAL-LOGIC` que implementa la lógica modal de Hennessy-Milner, en la Sección 4.7.3.

A continuación, definimos la *teoría* parámetro que especifica los requisitos que el parámetro debe satisfacer: debe tener una constante (`MOD`) que represente el módulo con las reglas de reescritura que serán utilizadas para construir el árbol de búsqueda, y una constante (`labels`) que represente la lista de etiquetas de estas reglas.

```
(fth AMODULE is
   including META-LEVEL .
   op MOD : -> Module .
   op labels : -> QidList .
 endfth)
```

El módulo con la estrategia, que extiende al módulo predefinido `META-LEVEL`, es entonces el módulo parametrizado `SEARCH[M :: AMODULE]`.

Ya que estamos definiendo una estrategia de búsqueda en un árbol de posibles reescrituras, necesitamos la noción de *objetivo* de la búsqueda. Para que la estrategia sea suficientemente general, supondremos que el módulo metarrepresentado `MOD` tiene una operación `ok` definida al nivel objeto (véase, por ejemplo, la Sección 4.3.3), la cual devuelve un valor del tipo `Answer` de modo que

- `ok(T) = solution` significa que el término `T` es uno de los términos que estamos buscando, es decir, `T` denota una solución;
- `ok(T) = no-solution` significa que el término `T` no es una solución y que no se puede encontrar ninguna solución por debajo de él en el árbol de búsqueda;
- `ok(T) = maybe-sol` significa que el término `T` no es una solución pero que no se sabe si hay soluciones por debajo de él.

La estrategia controla las posibles reescrituras de un término por medio de la función de descenso `meta-apply`, en el módulo `META-LEVEL`. Como se vio en la Sección 2.2.4, la evaluación de `meta-apply(MOD, T, L, S, N)` aplica (descartando los `N` primeros encajes) la regla del módulo `MOD` con etiqueta `L`, parcialmente instanciada con la sustitución `S`, al término `T` (al nivel más alto). La función devuelve el término resultado completamente reducido y la representación de la sustitución utilizada en la reducción, `{ T', S' }`, como un término de tipo `ResultPair` construido utilizando el operador `{_,_}`.

En la Sección 4.3.1 vimos la necesidad de instanciar las variables nuevas en la parte derecha de una regla de reescritura con el fin de crear nuevas metavariables. Si queremos utilizar la función `meta-apply` con reglas que tengan nuevas variables en la parte derecha (como `NEW1` en algunas de las reglas de la Sección 4.3.1), tenemos que proporcionar una sustitución de forma que las reglas se apliquen siempre sin variables nuevas en la parte derecha. Para simplificar la explicación, supondremos que una regla puede tener como mucho tres nuevas variables, llamadas `NEW1`, `NEW2` y `NEW3`. (Sin embargo, la versión dada en [VMO00a] y en la página web http://dalila.sip.ucm.es/~alberto/tesis es completamente general, admitiendo cualquier número de metavariables, determinado por una constante `num-metavar` de la teoría `AMODULE`.) Estas variables serán sustituidas por nuevos identificadores, formados por una comilla y un número, no utilizados anteriormente en la construcción del árbol conceptual de reescrituras actual.

```
vars N M : MachineInt .
op new-var : MachineInt -> Term .
eq new-var(M) = {conc('' , index(' , M))}'Qid .

op createSubst : MachineInt -> Substitution .
eq createSubst(M) = (('NEW1@Qid <- new-var(M + 1));
                     ('NEW2@Qid <- new-var(M + 2));
                     ('NEW3@Qid <- new-var(M + 3))) .
```

Por ejemplo, la evaluación de `new-var(5)` devuelve la metarrepresentación `{''5}'Qid` del término `'5` de tipo `Qid`. Para evitar conflictos con los nombres de las variables metarrepresentadas, Full Maude las renombra siguiendo el convenio de añadir el carácter '`@`' seguido del nombre del tipo de la variable. Así, por ejemplo, una variable metarrepresentada como `'NEW1` de tipo `Qid` se renombra a `'NEW1@Qid`. La evaluación de `createSubst(5)` devuelve la sustitución[4]

```
'NEW1@Qid <- {''6}'Qid ; 'NEW2@Qid <- {''7}'Qid ; 'NEW3@Qid <- {''8}'Qid
```

utilizada para reemplazar las variables `NEW1`, `NEW2` y `NEW3` por nuevos identificadores con comilla. En este caso, 5 sería el último número utilizado para construir identificadores nuevos en el proceso actual de reescritura, por lo que los tres identificadores creados son realmente *nuevos*.

Definimos ahora una nueva operación `meta-apply'` que recibe el mayor número `M` utilizado para sustituir variables en `T` y utiliza tres nuevos enteros para crear otros tantos nuevos identificadores (metarrepresentados).

[4] La sintaxis para construir sustituciones puede encontrarse en la Sección 2.2.4.

```
var L : Qid .
var T : Term .
var SB : Substitution .

op extTerm : ResultPair -> Term .
eq extTerm({T, SB}) = T .

op meta-apply' : Term Qid MachineInt MachineInt -> Term .
eq meta-apply'(T, L, N, M) = extTerm(meta-apply(MOD, T, L,
                                                createSubst(M), N)) .
```

La operación `meta-apply'` devuelve *una* de las posibles reescrituras en un paso, y al nivel más alto, del término recibido como parámetro. Nuestro primer paso consiste en definir una operacion `allRew` que devuelva *todas* las posibles reescrituras *secuenciales en un paso* [Mes92] del término `T` recibido como parámetro, utilizando para ello reglas de reescritura con etiquetas en la lista `labels`. El tercer argumento de `allRew` representa el mayor número `M` utilizado para crear nuevas variables en `T`.

El módulo `META-LEVEL` contiene un tipo `TermList` para construir listas de términos, pero no contiene ningún elemento identidad, que nosotros necesitamos ahora para representar el caso en el que ninguna regla pueda aplicarse, por lo que el conjunto de todas las reescrituras sería vacío. Por tanto extendemos el módulo de la siguiente forma:

```
op ~ : -> TermList .   *** empty term list
var TL : TermList .
eq ~, TL = TL .
eq TL, ~ = TL .
```

Obsérvese que este nuevo operador para representar la lista vacía de términos no funcionará, en principio, como elemento identidad del operador `_,_` de concatenación, ya que este ya está declarado en el módulo predefinido `META-LEVEL`, por lo que el sistema no hará encaje de patrones módulo identidad. Ahora bien, podemos añadir las dos ecuaciones indicadas, que se podrán utilizar, como cualquier otra ecuación, de izquierda a derecha, para simplificar términos de tipo `TermList` donde aparezca la constante de lista vacía `~`.

Las operaciones necesarias para encontrar todas las posibles reescrituras de un término, y sus definiciones, son las siguientes:

```
op allRew : Term QidList MachineInt  -> TermList .
op topRew : Term Qid MachineInt MachineInt -> TermList .
op lowerRew : Term Qid MachineInt -> TermList .
op rewArgs : Qid TermList TermList Qid MachineInt -> TermList .
op rebuild : Qid TermList TermList TermList -> TermList .

var LS : QidList .
vars C S OP : Qid .
vars Before After : TermList .

eq allRew(T, nil, M) = ~ .
```

```
eq allRew(T, L LS, M) = topRew(T, L, 0, M), *** rew. at the top
                        lowerRew(T, L, M),  *** rew. subterms
                        allRew(T, LS, M) .  *** rew. with labels LS
```

La evaluación de `topRew(T, L, N, M)` devuelve todas las posibles reescrituras en un paso, y al nivel más alto, del término `T`, aplicando la regla `L`, descartando los `N` primeros encajes, y utilizando números a partir de `M + 1` para construir identificadores para las variables nuevas.

```
eq topRew(T, L, N, M) =
    if meta-apply'(T, L, N, M) == error* then ~
    else (meta-apply'(T, L, N, M) , topRew(T, L, N + 1, M))
    fi .
```

La evaluación de `lowerRew(T, L, M)` devuelve todas las posibles reescrituras en un paso de los subtérminos de `T` aplicando la regla `L` y utilizando números a partir de `M + 1` para construir identificadores para las variables nuevas. Si `T` es una constante (término sin argumentos), se devuelve la lista vacía de términos; en caso contrario, se utiliza la operación `rewArgs`, definida de forma recursiva sobre su tercer argumento, el cual representa los argumentos de `T` aún no reescritos.

```
eq lowerRew({C}S, L, M) = ~ .
eq lowerRew(OP[TL], L, M) = rewArgs(OP, ~, TL, L, M) .

eq rewArgs(OP, Before, T, L, M) =
    rebuild(OP, Before, allRew(T, L, M), ~) .
eq rewArgs(OP, Before, (T, After), L, M) =
    rebuild(OP, Before, allRew(T, L, M), After) ,
    rewArgs(OP, (Before, T), After, L, M) .
```

La evaluación de `rebuild(OP, Before, TL, After)` devuelve todos los términos de la forma `OP[Before, T, After]` donde `T` es un término de la lista de términos `TL`. Los términos construidos de esta manera son metarreducidos, utilizando las ecuaciones en el módulo `MOD`, antes de ser devueltos.

```
eq rebuild(OP, Before, ~, After) = ~ .
eq rebuild(OP, Before, T, After) =
    meta-reduce(MOD, OP[Before, T, After]) .
eq rebuild(OP, Before, (T, TL), After) =
    meta-reduce(MOD, OP[Before, T, After]),
    rebuild(OP, Before, TL, After) .
```

Por ejemplo, podemos aplicar la operación `allRew` a la metarrepresentación del término `{{ 'a.0 + 'b.0 -- 'b -> 0 }}` para calcular todas sus reescrituras, obteniendo la metarrepresentación de los términos `{{ 'a.0 -- 'b -> 0 }}` y `{{ 'b.0 -- 'b -> 0 }}`.[5]

[5] En [VMO00a, Apéndice B] se pueden encontrar estos ejemplos tal y como se introducirían en el sistema

```
Maude> (reduce allRew( {{ 'a . 0 + 'b . 0 -- 'b -> 0 }}, labels, 0 ) .)
result TermList : {{ 'a . 0 -- 'b -> 0 }}, {{ 'b . 0 -- 'b -> 0 }}
```

Ahora ya estamos en condiciones de definir una estrategia para buscar en el árbol (conceptual) de posibles reescrituras de un término `T` algún término que satisfaga el predicado `ok`. Cada nodo del árbol de búsqueda es un par, cuya primera componente es un término y cuya segunda componente es un número que representa el mayor valor utilizado como identificador para nuevas variables en el proceso de reescritura del término. Los nodos del árbol que han sido generados pero que todavía no han sido comprobados se mantienen en una lista.

```
sorts Pair PairSeq .
subsort Pair < PairSeq .
op <_‘,_> : Term MachineInt -> Pair .
op nil : -> PairSeq .
op _|_ : PairSeq PairSeq -> PairSeq [assoc id: nil] .
var PS : PairSeq .
```

Necesitamos una operación para construir estos pares a partir de la lista de términos devuelta por `allRew`:

```
op buildPairs : TermList MachineInt -> PairSeq .
eq buildPairs(~, N) = nil .
eq buildPairs(T, N) = < T , N > .
eq buildPairs((T, TL), N) = < T , N > | buildPairs(TL, N) .
```

La operación `rewDepth` comienza la búsqueda llamando a la operación `rewDepth'` con la raíz del árbol de búsqueda. `rewDepth'` devuelve la primera solución encontrada en una búsqueda en profundidad. Si no hay ninguna solución, devuelve el término `error*`.

```
op rewDepth : Term -> Term .
op rewDepth' : PairSeq -> Term .

eq rewDepth(T) = rewDepth'(< meta-reduce(MOD, T), 0 >) .
eq rewDepth'(nil) = error* .
eq rewDepth'(< T , N > | PS) =
     if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then T
     else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer then
               rewDepth'(PS)
          else rewDepth'(buildPairs(allRew(T, labels, N), N + 3) | PS )
          fi)
     fi .
```

Full Maude. Aquí utilizamos $\overline{t}$ para la metarrepresentación del término t para simplificar la presentación. También podemos evitar la necesidad de introducir términos metarrepresentados si utilizamos las capacidades de análisis sintáctico y escritura de términos que proporciona Maude. Como el lenguaje CCS es suficientemente simple, no utilizamos estas capacidades en este capítulo, pero en cambio sí que las vamos a utilizar en la implementación de LOTOS en el Capítulo 7.

Hemos definido una estrategia de búsqueda al metanivel para resolver el "problema" de la aplicación no determinista de las reglas de reescritura. Este no determinismo es parte esencial de la lógica de reescritura. Maude 2.0 (véase la Sección 2.4, [CDE+00b]) incluye un comando `search` para buscar, en el árbol de todas las posibles reescrituras de un término, los términos que encajen con un patrón dado. Esto tiene la ventaja indiscutible de la eficiencia, ya que la búsqueda está integrada en la implementación del sistema[6]. Sin embargo, el definir la búsqueda al metanivel tiene la ventaja de permitirnos hacerla más general. Por ejemplo, aquí hemos utilizado un predicado `ok` definido por el usuario que define los objetivos de búsqueda. También hemos podido incluir la generación de nuevas metavariables en el proceso de búsqueda. De hecho, es la necesidad de instanciación de estas metavariables lo que no nos permitiría utilizar el nuevo comando `search` en esta representación, ya que este comando no puede utilizar directamente reglas de reescritura con variables nuevas en la parte derecha. Si adaptáramos la implementación presentada en este capítulo a la nueva versión de Maude, seguiría siendo necesaria la estrategia de búsqueda extendida con la generación de nuevas variables.

4.3.3. Algunos ejemplos

Ahora podemos probar la semántica de CCS con algunos ejemplos que manejan diferentes juicios. Primero definimos un módulo `CCS-OK` que extiende la sintaxis y reglas semánticas de CCS para definir algunos procesos constantes que se utilizarán en los ejemplos. También definimos la constante `context` utilizada por la semántica, y que contiene la definición de identificadores de procesos. Y por último definimos el predicado `ok`, que establece cuándo una configuración es una solución. En este caso una configuración denota una solución cuando es el conjunto vacío de juicios, representando el hecho de que el conjunto de juicios original es demostrable por medio de las reglas semánticas.

```
(mod CCS-OK is
   protecting CCS-SEMANTICS .

   ops p1 p2 p3 : -> Process .

   eq p1 = ('a . 0) + ('b . 0 | ('c . 0 + 'd . 0)) .
   eq p2 = ('a . 'b . 0 | (~ 'c . 0) ['a / 'c]) \ 'a .
   eq context = ('Proc =def 'a . tau . 'Proc) .
   eq p3 = ('Proc | ~ 'a . 'b . 0) \ 'a .

   sort Answer .

   ops solution  no-solution  maybe-sol : -> Answer .
   op ok : Configuration -> Answer .

   var JS : JudgementSet .
```

[6]Nosotros utilizaremos este comando de búsqueda y su representación en el metanivel en la implementación de CCS del Capítulo 5 y de LOTOS en el Capítulo 7.

```
   eq ok({{ emptyJS }}) = solution .
   ceq ok({{ JS }}) = maybe-sol if JS =/= emptyJS .
 endm)
```

Para instanciar el módulo genérico `SEARCH` necesitamos la metarrepresentación del módulo `CCS-OK`. Utilizamos la función `up` de Full Maude (véase Sección 2.3.3) para obtener la metarrepresentación de un módulo o un término.

```
(mod META-CCS is
   including META-LEVEL .
   op METACCS : -> Module .
   eq METACCS = up(CCS-OK) .
 endm)
```

Declaramos una vista (véase Sección 2.3.3) para instanciar el módulo genérico parametrizado `SEARCH`.

```
(view ModuleCSS from AMODULE to META-CCS is
   op MOD to METACCS .
   op labels to ('bind 'pref 'sum 'par 'res 'dist 'rel 'def) .
 endv)

(mod SEARCH-CCS is
  protecting SEARCH[ModuleCCS] .
 endm)
```

Ahora podemos ver ya los ejemplos. Primero probamos que el proceso `p1` puede realizar la acción `'c` convirtiéndose en `'b.0 | 0`.

```
Maude> (red rewDepth({{ p1 -- 'c -> 'b . 0 | 0 }}) .)
result Term : {{ emptyJS }}
```

También podemos probar que el proceso `p2` no puede realizar la acción `'a` (pero véase el comentario más abajo).

```
Maude> (red rewDepth({{ p2 -- 'a -> ('b . 0 | (~ 'c . 0)['a / 'c]) \ 'a }}) .)
result Term : error*
```

El hecho de que se haya devuelvo `error*` significa que en el árbol de reescrituras que se ha explorado no hay ningún término que satisfaga el predicado `ok`, es decir, no hay ningún término que represente la lista vacía de juicios. Por tanto, la transición dada

```
p2 -- 'a -> ('b . 0 | (~ 'c . 0)['a / 'c]) \ 'a
```

no es válida en CCS.

El proceso `p2` sí puede realizar la acción `tau` convirtiéndose en `('b.0 | 0['a/'c])\'a`.

```
Maude> (red rewDepth({{ p2 -- tau -> ('b . 0 | 0 ['a / 'c]) \ 'a }}) .)
result Term : {{ emptyJS }}
```

De la misma forma, podemos comprobar que el proceso `p3` puede realizar la acción `tau`.

```
Maude> (red rewDepth({{ p3 -- tau -> (tau . 'Proc | 'b . 0) \ 'a }}) .)
result Term : {{ emptyJS }}
```

En todos estos ejemplos hemos tenido que proporcionar el proceso resultado. En las demostraciones positivas no hay problema (aparte de la molestia de tener que escribir explícitamente este proceso), pero en la demostración negativa, es decir, cuando hemos visto que el proceso `p2` no puede realizar la acción `'a`, la demostración dada no es en realidad completa: lo que hemos probado es que el proceso `p2` no puede realizar la acción `'a` *convirtiéndose* en `('b.0 | (~'c.0) ['a/'c])\'a`, pero no hemos probado que no haya ninguna forma en la cual el proceso `p2` pueda ejecutar la acción `'a`. En la Sección 4.4.2 veremos cómo puede probarse correctamente tal cosa.

La corrección de las demostraciones obtenidas depende de la corrección de la estrategia de búsqueda. Por ejemplo, si la estrategia fuera incompleta, debido a no aplicar en cierto momento una regla de reescritura aplicable, la "demostración" de que un proceso no puede realizar una transición no sería válida. También las respuestas positivas podrían ser incorrectas, por ejemplo si la estrategia perdiera en algunos casos juicios que tienen que ser demostrados.

La simplicidad y la generalidad de la estrategia de búsqueda permiten evitar estos problemas. Una vez que contamos con una estrategia correcta que realmente construye y recorre completamente el árbol conteniendo todas las reescrituras de un término, entonces tenemos una estrategia correcta válida para siempre, que no tiene que ser modificada cuando modificamos la representación de la semántica subyacente, o incluso si cambiamos la semántica completamente, pasando a considerar otros lenguajes.

4.4. Cómo obtener nuevas clases de resultados

Ahora estamos interesados en responder a preguntas tales como: ¿Puede el proceso P realizar la acción a (sin preocuparnos de en qué se convierte el proceso al hacerlo)? Es decir, queremos saber si la transición $P \xrightarrow{a} P'$ es posible para un P' cualquiera o indeterminado. Este es el mismo problema que teníamos cuando aparecían nuevas variables en las premisas de una regla semántica. La solución, al igual que se hizo en el caso de las acciones, consiste en introducir metavariables para denotar procesos.

4.4.1. Inclusión de metavariables como procesos

Igual que hicimos con las acciones, declaramos un nuevo tipo de metavariables como procesos,

```
sort MetaVarProc .
op ?‘(_‘)P : Qid -> MetaVarProc .
```

y un nuevo tipo de posibles procesos,

```
sort Process? .
subsorts Process MetaVarProc < Process? .
var ?P : MetaVarProc .
var P? : Process? .
```

y modificamos la operación para construir los juicios básicos

```
op _--_->_ : Process? Act? Process? -> Judgement [prec 50] .
```

Ahora para el operador de prefijo tenemos que añadir dos nuevas reglas:

```
rl [pref] :  A . P -- A -> ?P
        => --------------------
                [?P := P] .

rl [pref] :   A . P -- ?A -> ?P
        => ------------------------
             [?A := A] [?P := P] .
```

donde, como en el caso de las metavariables como acciones, tenemos que definir una nueva clase de juicios que ligan metavariables como procesos, una regla para propagar estas ligaduras, y operaciones que realicen la correspondiente sustitución:

```
op ‘[_:=_‘] : MetaVarProc Process? -> Judgement .

rl [bind] : {{ [?P := P] JS }} => {{ <proc ?P := P > JS }} .

op <proc_:=_>_ : MetaVarProc Process Process? -> Process?  .
op <proc_:=_>_ : MetaVarProc Process Judgement -> Judgement .
op <proc_:=_>_ : MetaVarProc Process JudgementSet -> JudgementSet .
```

Para el resto de operadores CCS se tienen que añadir nuevas reglas que traten las metavariables en el segundo proceso de un juicio que represente una transición (véase el conjunto completo de reglas en [VMO00a] o en la página web). Aquí mostraremos solo algunos ejemplos de cómo son estas reglas.

```
rl [sum] :   P + Q -- A? -> ?P
        => --------------------
               P -- A? -> ?P .

rl [sum] :   P + Q -- A? -> ?Q
        => --------------------
               Q -- A? -> ?Q .

rl [par] :              P | Q -- A? -> ?P
        => ------------------------------------------
            P -- A? -> ?(NEW1)P  [?P := ?(NEW1)P | Q] .
```

```
rl [par] :                  P | Q -- ?A -> ?R
     => ------------------------------------------------------
         P -- ?(NEW1)A -> ?(NEW2)P   Q -- ~ ?(NEW1)A -> ?(NEW3)P
               [?A := tau] [?R := ?(NEW2)P | ?(NEW3)P] .

rl [res] :           P \ L -- ?A -> ?P
       => ----------------------------------------
           P -- ?A -> ?(NEW1)P  [?A =/= L] [?A =/= ~ L]
                     [?P := ?(NEW1)P \ L] .
```

4.4.2. Más ejemplos

Ahora podemos probar que el proceso `p1` puede realizar la acción `'c` reescribiendo el juicio

`p1 -- 'c -> ?('any)P`,

donde, al colocar la metavariable `?('any)P` como segundo proceso, indicamos que no nos preocupa el proceso resultante. Obviamente, el nombre concreto utilizado en esta metavariable (`'any`) es irrelevante.

```
Maude> (red rewDepth(~~{{ p1 -- 'c -> ?('any)P }}~~) .)
result Term : ~~{{ emptyJS }}~~
```

También podemos probar que el proceso `p2` *no puede* realizar la acción `'a`.

```
Maude> (red rewDepth(~~{{ p2 -- 'a -> ?('any)P }}~~) .)
result Term : error*
```

4.4.3. Sucesores de un proceso

Otra pregunta interesante es saber cuáles son los *sucesores* de un proceso P tras realizar una acción en un conjunto dado As. Dicho conjunto viene dado por

$$succ(P, As) = \{P' \mid P \xrightarrow{a} P' \wedge a \in As\}.$$

Ya que tenemos metavariables como procesos, podemos reescribir el juicio que representa una transición

`P -- A -> ?('proc)P`

instanciando la variable `A` con las acciones en el conjunto de acciones dado. Estas reescrituras ligarán la metavariable `?('proc)P` con los sucesores de `P`, pero nos encontramos entonces con dos problemas. El primero es que, con la implementación actual, las ligaduras entre metavariables y procesos se pierden cuando se realiza la sustitución aplicando la regla de reescritura `bind`. Para resolver este problema modificamos el operador que construye configuraciones, para que mantenga, junto con el conjunto de juicios que deben ser probados, el conjunto de ligaduras producidas hasta el momento:

```
op `{`{_|_`}`} : JudgementSet JudgementSet -> Configuration .
```

La regla `bind` guardará en este segundo argumento las ligaduras generadas además de realizar la correspondiente propagación:

```
rl [bind] : {{ [?P := P] JS | JS' }} =>
            {{ (<proc ?P := P > JS) | [?P := P] JS' }} .
```

También tenemos que modificar la definición de la operación `ok`. Ahora una configuración es solución si su primera componente representa el conjunto vacío de juicios, independientemente del valor de su segunda componente:

```
vars JS JS' : JudgementSet .
eq ok({{ emptyJS | JS' }}) = solution .
ceq ok({{ JS | JS'}}) = maybe-sol if JS =/= emptyJS .
```

El otro problema es que la función `rewDepth` solo devuelve una solución, pero podemos modificarla fácilmente de forma que consigamos *todas* las soluciones, es decir, de forma que se explore (en profundidad) el árbol completo de posibles reescrituras, encontrando *todos* los nodos que satisfagan el predicado `ok`. La operación `allSol`, añadida al módulo `SEARCH`, devuelve el conjunto con los términos que representan todas las soluciones. Añadimos también un tipo `TermSet` para representar conjuntos de términos metarrepresentados. Nótese cómo se introduce la "idempotencia" mediante una ecuación: dos términos se consideran iguales cuando lo son en el nivel objeto, es decir, en el módulo `MOD`.

```
sort TermSet .
subsort Term < TermSet .

op `{`} : -> TermSet .
op _U_ : TermSet TermSet -> TermSet [assoc comm id: {}] .
ceq T U T' = T if meta-reduce(MOD, '_==_[T, T']) == {'true}'Bool .

op allSol : Term -> TermSet .
op allSolDepth : PairSeq -> TermSet .

eq allSol(T) = allSolDepth(< meta-reduce(MOD,T), 0 >) .
eq allSolDepth(nil) = {} .
eq allSolDepth(< T , N > | PS) =
   if meta-reduce(MOD, 'ok[T]) == {'solution}'Answer then
      (T U allSolDepth(PS))
   else (if meta-reduce(MOD, 'ok[T]) == {'no-solution}'Answer then
            allSolDepth(PS)
         else
            allSolDepth(buildPairs(allRew(T, labels, N), N + 3) | PS)
         fi)
   fi .
```

Ahora podemos definir, en una extensión de SEARCH-CCS, una operación succ que, dada la metarrepresentación de un proceso y un conjunto de metarrepresentaciones de acciones, devuelva el conjunto de metarrepresentaciones de los procesos sucesores[7]

```
op succ : Term TermSet -> TermSet .

eq succ(T, {}) = {} .
eq succ(T, A U AS) =
       filter(allSol({{ T -- A -> ?('proc)P | emptyJS }}), ?('proc)P)
     U succ(T, AS) .
```

donde se utiliza la función filter para eliminar todas las ligaduras que tengan que ver con metavariables distintas a ?('proc)P.

El módulo CCS-SUCC que implementa la extensión de SEARCH-CCS es el siguiente:

```
(fmod CCS-SUCC is
  protecting SEARCH-CCS .

  op succ : Term TermSet -> TermSet .
  op filter : TermSet TermSet -> TermSet .
  op filter-bind : TermList TermSet -> TermSet .

  var A T T' BS : Term .
  vars MVS TS AS : TermSet .
  var TL : TermList .

  eq filter({}, MVS) = {} .
  eq filter(_U_( ''{'{_|_'}'}[{'emptyJS}'JudgementSet, BS], TS), MVS) =
       _U_(filter-bind(BS, MVS), filter(TS, MVS)) .
  eq filter-bind( ('`[_:=_`] [T, T']) , MVS) =
       if (T isIn MVS) then T' else {} fi .
  eq filter-bind( ('`[_:=_`] [T, T'], TL) , MVS) =
       if (T isIn MVS) then _U_(T', filter-bind(TL, MVS))
       else filter-bind(TL, MVS)
       fi .
  eq filter-bind('__ [ TL ] , MVS) = filter-bind(TL, MVS) .

  eq succ(T, {}) = {} .
  eq succ(T, A U AS) =
       filter(allSol(''{'{_|_'}'}['_--_->_ [T,A,'?'(_')P [{''proc}'Qid]],
                                  {'emptyJS}'JudgementSet]),
              '?'(_')P [{''proc}'Qid])
     U succ(T,AS) .
endfm)
```

Con los procesos definidos en el módulo CCS-OK de la Sección 4.3.3 podemos ilustrar estas operaciones. Por ejemplo, podemos calcular los sucesores del proceso p1 tras realizar bien la acción 'a o la acción 'b.

[7] En esta definición también utilizamos $\overline{t}$ para denotar la metarrepresentación del término t.

$$\frac{}{P \xrightarrow{\varepsilon} P} \qquad \frac{P \xrightarrow{a_1} P' \quad P' \xrightarrow{a_2 \ldots a_n} P''}{P \xrightarrow{a_1 a_2 \ldots a_n} P''}$$

Figura 4.2: Semántica de trazas.

```
Maude> (red succ(p1, 'a U 'b) .)
result TermSet : 0 | ( 'c.0 + 'd.0 ) U 0
```

4.5. Extensión de la semántica a trazas

En la semántica de CCS que hemos utilizado hasta ahora solo aparecían transiciones de la forma $P \xrightarrow{a} P'$. Podemos extender la semántica a secuencias de acciones o *trazas*, es decir, a transiciones de la forma $P \xrightarrow{a_1 a_2 \ldots a_n} P'$, con $n \geq 0$. Las reglas semánticas que definen estas nuevas transiciones se muestran en la Figura 4.2, donde ε denota la traza vacía.

Al efecto comenzamos extendiendo nuestro entorno incorporando un nuevo tipo de datos para representar las trazas

```
sort Trace .
subsort Act < Trace .
op nil : -> Trace .
op __ : Trace Trace -> Trace [assoc id: nil] .
var Tr : Trace .
```

y una nueva clase de juicios

```
op _-_->_ : Process Trace Process -> Judgement [prec 50] .
```

Y ahora podemos traducir las reglas semánticas presentadas en la Figura 4.2 a las correspondientes reglas de reescritura, de la forma obvia:

```
rl [nil] :     P - nil -> P
        => ------------------
                emptyJS .

rl [seq] :             P - A Tr -> P'
        => ----------------------------------------
            P -- A -> ?(NEW1)P   ?(NEW1)P - Tr -> P' .
```

Igual que hicimos con los juicios sobre acciones, podemos extender los juicios sobre trazas de forma que se puedan utilizar metavariables como procesos. Solo tenemos que modificar la operación para construir esta clase de juicios

```
op _-_->_ : Process? Trace Process? -> Judgement [prec 50] .
```

$$\frac{P \xrightarrow{\tau}{}^{*} Q \quad Q \xrightarrow{a} Q' \quad Q' \xrightarrow{\tau}{}^{*} P'}{P \overset{a}{\Longrightarrow} P'}$$

$$\frac{}{P \xrightarrow{a}{}^{*} P} \qquad \frac{P \xrightarrow{a} P' \quad P' \xrightarrow{a}{}^{*} P''}{P \xrightarrow{a}{}^{*} P''}$$

$$\frac{P \xrightarrow{\tau}{}^{*} P'}{P \overset{\varepsilon}{\Longrightarrow} P'} \qquad \frac{P \overset{a_1}{\Longrightarrow} P' \quad P' \overset{a_2 \ldots a_n}{\Longrightarrow} P''}{P \overset{a_1 a_2 \ldots a_n}{\Longrightarrow} P''}$$

Figura 4.3: Semántica débil.

y añadir nuevas reglas para tratar el caso en el que una metavariable aparezca como segundo proceso

```
rl [nil] :  P - nil -> ?P
         => ---------------
              [?P := P] .

rl [seq] :            P - A Tr -> ?P
         => ----------------------------------------
             P -- A -> ?(NEW1)P   ?(NEW1)P - Tr -> ?P .
```

4.6. Extensión a la semántica de transiciones débil

Otra relación de transición importante definida para CCS, $P \overset{a}{\Longrightarrow} P'$, no observa las transiciones τ [Mil89]. Se define en la Figura 4.3, donde $\xrightarrow{\tau}{}^{*}$ denota el cierre reflexivo y transitivo de $\xrightarrow{\tau}$, definiéndose también para una acción a cualquiera como se indica en la segunda fila de la Figura 4.3. La semántica de transición débil se puede extender también a trazas, como muestra la tercera fila de la Figura 4.3.

4.6.1. Definición de la extensión

En esta sección veremos cómo se definen en Maude estas extensiones. Primero definimos el cierre reflexivo y transitivo de la relación de transición básica que ya hemos implementado. Para ello necesitamos un operador para construir la nueva clase de juicios

```
op _--_->*_ : Process? Act Process? -> Judgement [prec 50] .
```

donde se permiten metavariables como procesos, porque así se necesitan en la implementación de la relación de transición débil. Las reglas de reescritura que definen el cierre son las siguientes:

```
rl [refl] :   P -- A ->* P
         => -----------------
               emptyJS .

rl [refl] :  P -- A ->* ?P
         => ----------------
              [?P := P] .

rl [tran] :                P -- A ->* P?
         => ------------------------------------------
             P -- A -> ?(NEW1)P   ?(NEW1)P -- A ->* P? .
```

Nótese que la transición básica $\xrightarrow{a}$ está incluida en su cierre transitivo y reflexivo $\xrightarrow{a}{}^*$, lo que se obtiene utilizando primero la regla de reescritura `tran` y aplicando después una de las reglas `refl` a la segunda premisa.

De la misma forma podemos definir en Maude la relación de transición débil, tanto para acciones como para trazas:

```
op _==_==>_ : Process? Act Process? -> Judgement [prec 50] .

rl [act] :                 P == A ==> P?
        => ---------------------------------------------------
            P -- tau ->* ?(NEW1)P   ?(NEW1)P -- A -> ?(NEW2)P
                        ?(NEW2)P -- tau ->* P? .


op _=_==>_ : Process? Trace Process? -> Judgement [prec 50] .

rl [nil] :    P = nil ==> P?
        => --------------------
              P -- tau ->* P? .

rl [seq] :                 P = A Tr ==> P?
        => ------------------------------------------
            P == A ==> ?(NEW1)P   ?(NEW1)P = Tr ==> P? .
```

Como antes, podemos definir una función que calcule los sucesores de un proceso con respecto a la semántica de transiciones débiles:

$$wsucc(P, As) = \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in As\}.$$

```
op wsucc : Term TermSet -> TermSet .

eq wsucc(T, {}) = {} .
eq wsucc(T, A U AS) = filter(
     allSol({{ T == A ==> ?('proc)P | emptyJS }}), ?('proc)P)
     U wsucc(T, AS) .
```

4.6.2. Ejemplo

Continuando con el ejemplo definido en el módulo `CCS-OK` en la Sección 4.3.3, podemos probar ahora que el proceso `p2` puede realizar una acción `'b` observable:

```
Maude> (red rewDepth({{ p2 == 'b ==> ?('any)P }}) .)
result Term : {{ emptyJS }}
```

También podemos probar que el proceso recursivo `'Proc` puede realizar sucesivamente tres acciones `'a` observables, convirtiéndose en el mismo proceso:

```
Maude> (red rewDepth({{ 'Proc = 'a 'a 'a ==> 'Proc }}) .)
result Term : {{ emptyJS }}
```

Finalmente, podemos calcular los sucesores del proceso `'Proc` definido recursivamente, después de realizar la acción `'a` permitiendo antes y después un número cualquiera de transiciones τ:

```
Maude> (red wsucc('Proc, 'a) .)
result TermSet : tau . 'Proc U 'Proc
```

4.7. Lógica modal para procesos CCS

En esta sección se muestra cómo se puede definir en Maude la semántica de una lógica modal para procesos CCS siguiendo el mismo enfoque que hemos usado para representar la semántica operacional de CCS, y utilizando las operaciones definidas en las secciones anteriores.

4.7.1. Lógica de Hennessy-Milner

Presentaremos aquí una lógica *modal* para describir capacidades locales de procesos CCS, que implementaremos en Maude en la siguiente sección. Esta lógica es una versión de la lógica de Hennessy-Milner [HM85] y su semántica se presenta en [Sti96]. Las fórmulas de la lógica se construyen de la siguiente manera:

$$\Phi ::= \texttt{tt} \mid \texttt{ff} \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K \rangle \Phi$$

donde K es un conjunto de acciones. La relación de satisfacción que describe cuando un proceso P satisface una propiedad Φ, $P \models \Phi$, se define en la Figura 4.4.

Tenemos que todo proceso satisface la fórmula `tt` y ninguno satisface la fórmula `ff`. Un proceso P satisface la fórmula $\Phi_1 \wedge \Phi_2$ si P satisface a la vez Φ_1 y Φ_2, y satisface la fórmula $\Phi_1 \vee \Phi_2$ si satisface Φ_1 o Φ_2. Un proceso P satisface la fórmula $[K]\Phi$ construida con el operador modal universal, si todos los sucesores en un paso de P tras realizar una acción en el conjunto K cumplen la formula Φ. En cambio, un proceso P satisface la fórmula $\langle K \rangle \Phi$ construida con el operador modal existencial, si alguno de sus sucesores cumple Φ.

$$
\begin{array}{ll}
P \models \mathtt{tt} & \\
P \models \Phi_1 \wedge \Phi_2 & \text{iff } P \models \Phi_1 \text{ and } P \models \Phi_2 \\
P \models \Phi_1 \vee \Phi_2 & \text{iff } P \models \Phi_1 \text{ or } P \models \Phi_2 \\
P \models [K]\Phi & \text{iff } \forall\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi \\
P \models \langle K \rangle \Phi & \text{iff } \exists\, Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi
\end{array}
$$

Figura 4.4: Relación de satisfacción de la lógica modal.

4.7.2. Implementación

Para implementar la lógica en Maude, primero definimos un tipo de datos `HMFormula` para representar fórmulas, y operadores que permitirán construir dichas fórmulas:

```
(mod MODAL-LOGIC is
  protecting CCS-SUCC .

  sort HMFormula .

  ops tt ff : -> HMFormula .
  op _/\_ : HMFormula HMFormula -> HMFormula .
  op _\/_ : HMFormula HMFormula -> HMFormula .
  op `[_`]_ : TermSet HMFormula -> HMFormula .
  op <_>_ : TermSet HMFormula -> HMFormula .
```

A continuación, definimos la semántica de la lógica modal de la misma manera que hicimos con la semántica de CCS, es decir, definiendo reglas de reescritura que reescriban el juicio $P \models \Phi$ al conjunto de juicios que tienen que ser satisfechos para que el primero se cumpla. Los tipos `Judgement` y `JudgementSet` tienen que volver a ser declarados, ya que en el módulo `CCS-SUCC` solo aparecen dentro de una constante de tipo `Module`, y metarrepresentados.

```
  sort Judgement .
  op _|=_ : Term HMFormula -> Judgement .

  sort JudgementSet .
  op emptyJS : -> JudgementSet .
  subsort Judgement < JudgementSet .
  op __ : JudgementSet JudgementSet -> JudgementSet [assoc comm id: emptyJS] .

  op _|=_ : Term HMFormula -> Judgement .
  op forall : TermSet HMFormula -> JudgementSet .
  op exists : TermSet HMFormula -> JudgementSet .

  var P : Term .
  vars K PS : TermSet .
  vars Phi Psi : HMFormula .
```

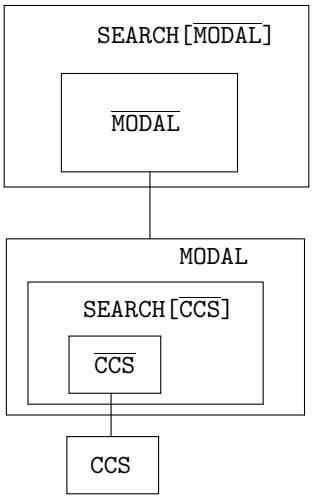


Figura 4.5: Estructura de módulos en nuestra aplicación.

```
  rl [true] : P |= tt          => emptyJS .
  rl [and]  : P |= Phi /\ Psi => (P |= Phi) (P |= Psi) .
  rl [or]   : P |= Phi \/ Psi => P |= Phi .
  rl [or]   : P |= Phi \/ Psi => P |= Psi .
  rl [box]  : P |= [ K ] Phi  => forall(succ(P, K), Phi) .
  rl [diam] : P |= < K > Phi  => exists(succ(P, K), Phi) .

  eq forall({}, Phi) = emptyJS .
  eq forall(P U PS, Phi) = (P |= Phi) forall(PS, Phi) .

  rl [ex] : exists(P U PS, Phi) => P |= Phi .
endm)
```

Estas reglas también son no deterministas. Por ejemplo, la aplicación de las dos reglas `or` es no determinista porque ambas tienen la misma parte izquierda, y la regla `ex` también es no determinista debido a los múltiples encajes con el patrón `P U PS`, módulo asociatividad y conmutatividad.

Podemos instanciar el módulo parametrizado `SEARCH` de la Sección 4.3.2 con la metarrepresentación del módulo que contiene la definición de la semántica de la lógica modal. Obtenemos así la estructura de módulos mostrada en la Figura 4.5. En la base tenemos el módulo `CCS` con la implementación de la semántica de CCS extendida con el predicado `ok` utilizado por la estrategia de búsqueda. Su metarrepresentación $\overline{\texttt{CCS}}$ se utiliza para instanciar la estrategia de búsqueda, y esta instanciación se utiliza para definir la semántica de la lógica modal de Hennessy-Milner en el módulo `MODAL`. Este módulo extendido con su propia definición del predicado `ok` es metarrepresentado, obteniéndose $\overline{\texttt{MODAL}}$, que se utiliza para instanciar de nuevo la estrategia de búsqueda.

4.7.3. Ejemplos

Al igual que hicimos en la Sección 4.3.3, para poder instanciar la estrategia de búsqueda ahora extendida, necesitamos definir un predicado `ok`, crear una vista e instanciar el módulo `SEARCH`.

```
(mod MODAL-LOGIC-OK is
  protecting MODAL-LOGIC .
  sort Answer .
  ops solution no-solution maybe-sol : -> Answer .
  op ok : JudgementSet -> Answer .
  var JS : JudgementSet .
  eq ok(emptyJS) = solution .
  ceq ok(JS) = maybe-sol if JS =/= emptyJS .
endm)

(mod META-MODAL-LOGIC-OK is
  including META-LEVEL .
  op METAMODALCCS : -> Module .
  eq METAMODALCCS = up(MODAL-LOGIC-OK) .
  op labelsIDs : -> QidList .
  eq labelsIDs = ( 'true 'and 'or 'box 'diam 'ex ) .
endm)

(view ModuleMODAL-LOGIC-OK from AMODULE to META-MODAL-LOGIC-OK is
  op MOD to METAMODALCCS .
  op labels to labelsIDs .
endv)

(mod SEARCH-MODAL-CCS is
  protecting SEARCH[ModuleMODAL-LOGIC-OK] .
endm)
```

Tomando como ejemplo uno de [Sti96], mostramos algunas fórmulas que cumple una máquina expendedora `'Ven` definida en un contexto CCS de la siguiente manera:

```
  eq context = 'Ven =def '2p . 'VenB  + '1p . 'VenL  &
               'VenB =def 'big . 'collectB . 'Ven    &
               'VenL =def 'little . 'collectL . 'Ven .
```

y cómo pueden ser probadas las mismas en Maude.

El proceso `'Ven` puede aceptar inicialmente una moneda del tipo `2p` o `1p`. Si se deposita una moneda `2p`, a continuación se puede presionar el botón `big`, y se puede recoger un producto grande. En cambio, si se deposita una moneda `1p`, se puede pulsar el botón `little`, y se puede recoger un producto pequeño. Después de ser recogido el producto, la máquina expendedora vuelve al estado inicial.

Una de las propiedades que cumple la máquina expendedora es que los botones no pueden ser pulsados inicialmente, es decir, antes de que se introduzca dinero. Podemos probar en Maude que `'Ven |= ['big, 'little]ff`:

```
Maude> (red rewDepth('Ven |= [ 'big U 'little ] ff) .)
result Term : emptyJS
```

Otra propiedad interesante de `'Ven` es que después de insertar una moneda `'2p`, el botón pequeño no puede pulsarse, mientras que el grande sí:

```
Maude> (red rewDepth('Ven |= ['2p](([']ittle] ff) /\ (<'big> tt))).)
result Term : emptyJS
```

`'Ven` también cumple que justo después de que se haya introducido una moneda no puede introducirse otra:

```
Maude> (red rewDepth('Ven |= [ '1p U '2p ] [ '1p U '2p ] ff) .)
result Term : emptyJS
```

Por último, también se cumple que después de introducir una moneda y pulsar el correspondiente botón, la máquina devuelve un producto (grande o pequeño):

```
Maude> (red rewDepth(
          'Ven |= ['1p U '2p]['big U 'little]< 'collectB U 'collectL > tt) .)
result Term : emptyJS
```

También podemos probar que un proceso no cumple una determinada fórmula. Por ejemplo, podemos ver que después de introducir una moneda `1p` no es posible presionar el botón `big` y recoger un producto grande.

```
Maude> (red rewDepth('Ven |= <'1p><'big>< 'collectB> tt) .)
result Term : error*
```

Obtener como resultado `error*` significa que la búsqueda no ha tenido éxito, es decir, que la fórmula dada *no* se puede probar con las reglas semánticas o, dicho de otra manera, es falsa.

4.7.4. Más modalidades

Si queremos dar a la acción "silenciosa" τ un estatus especial, podemos introducir nuevas modalidades $[\![K]\!]$ y $\langle\langle K\rangle\rangle$ definidas por medio de la relación de transición débil, en la Figura 4.6. Un proceso P satisface la fórmula $[\![K]\!]\Phi$ si todos sus sucesores alcanzables realizando una acción en el conjunto K según la semántica de transiciones débiles satisfacen la fórmula Φ. De igual modo, un proceso P satisface la fórmula $\langle\langle K\rangle\rangle\Phi$ si alguno de sus sucesores según dicha semántica cumple Φ. Obsérvese la analogía con los operadores modales anteriores, pues nos limitamos a cambiar el tipo de las transiciones para definir los sucesores de un proceso.

$$P \models [\![K]\!]\Phi \quad \text{iff } \forall Q \in \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in K\} \,.\, Q \models \Phi$$
$$P \models \langle\!\langle K \rangle\!\rangle \Phi \quad \text{iff } \exists Q \in \{P' \mid P \stackrel{a}{\Longrightarrow} P' \wedge a \in K\} \,.\, Q \models \Phi$$

Figura 4.6: Relación de satisfacción para los nuevos operadores modales.

La implementación en Maude de estas nuevas modalidades es la siguiente:

```
op `[`[_`]`]_ : TermSet HMFormula -> HMFormula .
op <<_>>_ : TermSet HMFormula -> HMFormula .

rl [dbox]  : P |= [[ K ]] Phi => forall(wsucc(P, K), Phi) .
rl [ddiam] : P |= << K >> Phi => exists(wsucc(P, K), Phi) .
```

donde se utiliza la operación `wsucc` para calcular los sucesores del proceso `P` con respecto a la semántica de transición débil. De nuevo la única diferencia con las reglas `box` y `diam` de la Sección 4.7.2 es la operación que se utiliza para calcular los sucesores de un proceso tras hacer acciones de un conjunto dado: `succ` pasa a ser `wsucc`.

Ahora podemos demostrar algunas propiedades que cumple un control de cruce entre una línea de ferrocarril y una carretera [Sti96], especificado en un contexto CCS como sigue:

```
eq context =
    'Road =def 'car . 'up . ~ 'ccross . ~ 'down . 'Road      &
    'Rail =def 'train . 'green . ~ 'tcross . ~ 'red . 'Rail  &
    'Signal =def  ~ 'green . 'red . 'Signal
                + ~ 'up . 'down . 'Signal                    &
    'Crossing =def ('Road | ('Rail | 'Signal)) \ 'green \ 'red \ 'up \ 'down .
```

El control está formado por tres componentes: `Road`, `Rail` y `Signal`. Las acciones `car` y `train` representan el acercamiento de un coche o un tren, respectivamente, `up` abre el cruce para el coche, $\overline{\texttt{ccross}}$ indica que el coche está pasando, `down` cierra el cruce para el coche, `green` representa la recepción de la señal verde por parte del tren, $\overline{\texttt{tcross}}$ representa que el tren está pasando, y `red` pone la luz roja.

El proceso `'Crossing` cumple que cuando un coche y un tren llegan simultáneamente al cruce, solo uno de ellos tiene la posibilidad de cruzarlo.

```
Maude> (red RewDepth(
   'Crossing |= [['car]][['train]]((<<~ 'ccross>> tt) \/ (<<~ 'tcross>> tt))) .)
result Term : emptyJS
Maude> (red RewDepth(
   'Crossing |= [['car]][['train]]((<<~ 'ccross>> tt) /\ (<<~ 'tcross>> tt))) .)
result Term : error*
```

4.8. Comparación con Isabelle

En esta sección se muestra cómo la semántica de CCS y la lógica modal de Hennessy-Milner se pueden representar y utilizar en el demostrador de teoremas Isabelle [NPW02], comparándose este marco con el nuestro.

Isabelle es un sistema genérico para implementar formalismos lógicos, e Isabelle/HOL es la especialización de Isabelle para la lógica de orden superior HOL (Higher-Order Logic) [NPW02].

Trabajar con Isabelle consiste en escribir teorías, las cuales son colecciones de tipos, funciones y teoremas. HOL contiene una teoría `Main`, unión de todas las teorías básicas predefinidas tales como la aritmética entera, las listas, los conjuntos, etc. Todas nuestras teorías extenderán esta. En una teoría hay tipos, términos y fórmulas HOL. Los términos se construyen como en programación funcional, aplicando funciones a argumentos, y las fórmulas son términos de tipo `bool`.

Isabelle distingue entre variables libres y ligadas, definidas en la forma habitual, y, además, tiene un tercer tipo de variables, llamadas *variables esquemáticas* o *desconocidas*, cuyo nombre empieza por `?`. Desde un punto de vista lógico, una variable desconocida es una variable libre, pero que durante el proceso de demostración puede ser instanciada mediante unificación con otro término. Las variables desconocidas corresponden a lo que nosotros hemos llamado *metavariables*.

Los tipos de datos definidos de forma inductiva forman parte de casi cualquier aplicación no trivial de HOL. Se introducen por medio de la palabra clave **datatype**, y se definen por medio de constructores. Por ejemplo, las acciones y los procesos CCS se modelan de forma natural con los siguientes tipos de datos:

```
types id = "char list"

datatype label = nm id
               | conm id

consts compl :: "label ⇒ label"
primrec "compl (nm a) = conm a"
        "compl (conm a) = nm a"

datatype action = tau
                | va label

datatype process = Stop                (("0")
       | Prefix action  process        ("_ . _" [120, 110] 110)
       | Sum    process process       (infixl "+" 85)
       | Par    process process       (infixl "|" 90)
       | Rel    process label label   ("_ [ _ =: _ ]" [100,120,120] 100)
       | Rest   process label         ("_ \ _" [100, 120] 100)
       | Id id
```

Las funciones sobre los tipos de datos (introducidas con la palabra clave **consts**) se

definen usualmente mediante *recursión primitiva*. La palabra clave **primrec** va seguida de la lista de ecuaciones. Así hemos definido más arriba la función `compl`. Utilizando todas estas herramientas podemos definir también los contextos de CCS.

```
datatype Def = Defi id process
datatype Context = Empty
                 | And Def Context

consts defin :: "id ⇒ Context ⇒ process"
primrec
  "defin X Empty = Stop"
  "defin X (And D C') =
    (case D of (Defi Y P) ⇒ (if (X = Y) then P else (defin X C')))"

consts Context :: Context
```

Para definir la semántica de CCS en Isabelle/HOL, definimos de forma inductiva un conjunto `trans` de ternas `(P,a,P')`. Utilizamos la notación mixfija habitual `P - a → P'` para representar que `(P,a,P') ∈ trans`. La inductividad implica que el conjunto de transiciones está formado exactamente por aquellas ternas `(P,a,P')` que pueden derivarse a partir del conjunto de reglas de transición. Los conjuntos definidos de forma inductiva se introducen en Isabelle por medio de una declaración **inductive**, que incluye una serie de reglas de introducción. Estas no son otra cosa que la traducción inmediata de las reglas semánticas de CCS. Cada regla de inferencia de la forma

$$\frac{\phi_1 \ \cdots \ \phi_n}{\phi}$$

se formaliza en la metalógica de Isabelle como el axioma $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$.

Aunque no utilicemos variables esquemáticas directamente en la regla, cuando se construye una teoría Isabelle transforma las reglas a una forma estándar, donde todas las variables libres se convierten en esquemáticas, lo que permite que la unificación instancie la regla.

A continuación, se presentan las reglas que definen la semántica de CCS en Isabelle:

```
inductive trans
intros
  prefi: "A . P - A → P"
  sum1:  "P1 - A → P ⟹ P1 + P2 - A → P"
  sum2:  "P2 - A → P ⟹ P1 + P2 - A → P"
  par1:  "P1 - A → P ⟹ P1 | P2 - A → P | P2"
  par2:  "P2 - A → P ⟹ P1 | P2 - A → P1 | P"
  par3:  "⟦ P1 - va L → P ; P2 - va (compl L) → Q ⟧
       ⟹  P1 | P2 - tau → P | Q"
  rel1:  "⟦ P - va N → P' ; N ≠ L ; N ≠ compl L ⟧
       ⟹  P [ M =: L ] - va N → P' [ M =: L ]"
  rel2:  "P - tau → P' ⟹ P [ M =: L ] - tau → P' [ M =: L ]"
  rel3:  "P - va L → P' ⟹   P [ M =: L ] - va M → P' [ M =: L ]"
```

```
rel4:  "⟦ P - va N → P' ; N = compl L ⟧
     ⟹  P [ M =: L ] - va (compl M) → P' [ M =: L ]"
res1:  "P - tau → P' ⟹ P \ L - tau → P' \ L"
res2:  "⟦ P - va L → P' ; L ≠ M ; compl L ≠ M ⟧
     ⟹ P \ M - va L → P' \ M"
defi:  "(defin X Context) - A → Q ⟹ Id X - A → Q"
```

La simplificación es una de las herramientas de demostración de teoremas más importante en Isabelle, y en otros muchos sistemas. En su forma más básica, la simplificación consiste en la aplicación repetida de ecuaciones de izquierda a derecha (reescritura de términos). Para facilitar la simplificación, los teoremas pueden declararse como reglas de simplificación con el atributo *[simp]*, en cuyo caso las demostraciones por simplificación harán uso de estas reglas automáticamente. Además las construcciones **datatype** y **primrec** declaran de forma oculta reglas de simplificación muy útiles.

Las demostraciones en Isabelle generalmente utilizan *resolución* para dar soporte a las demostraciones hacia atrás, donde se comienza con un objetivo que se va refinando progresivamente a subobjetivos más simples, hasta que todos han sido probados. El *resolutor clásico* de Isabelle está formado por una familia de herramientas que realizan demostraciones de forma automática. Podemos extender la potencia del resolutor añadiendo nuevas reglas (*intro:*). El resolutor clásico utiliza estrategias de búsqueda y vuelta atrás para *probar* un objetivo. En concreto, nosotros utilizaremos el método *force* que combina la aplicación del resolutor clásico con el simplificador a un objetivo.

Tal y como hicimos con Maude (Sección 4.3.3), probaremos la semántica de CCS con una serie de ejemplos. Para empezar, los procesos constantes se declaran por medio de definiciones (**defs**), que expresan abreviaturas y también serán aplicadas como reglas de simplificación.

```
consts p1 :: process
defs p1_def[simp]: "p1 ≡ ( va (nm ''a'') . 0) +
     (va (nm ''b'') . 0 | (va (nm ''c'') . 0 + va (nm ''d'') . 0))"

consts p2 :: process
defs p2_def[simp]: "p2 ≡ ( va (nm ''a'') . va (nm ''b'') . 0 |
     ( va (conm ''c'') . 0) [ nm ''a'' =: nm ''c''] ) \ nm ''a''"

lemma "p1 - va (nm ''c'') → va (nm ''b'') . 0 | 0"
by (force intro: trans.intros)
```

Podemos demostrar que un proceso puede realizar una acción dada, utilizando una variable desconocida *?P*, como hicimos en la Sección 4.4.2.

```
lemma "p2 - tau  → ?P"
by (force intro: trans.intros)
```

Pero para probar que una determinada transición *no* es posible, no podemos limitarnos a utilizar alguno de los métodos de demostración automáticos; por el contrario, tenemos

que utilizar inducción sobre la regla *trans*. Isabelle genera automáticamente una regla de eliminación para análisis por casos (regla *cases*) y una regla de inducción por cada definición inductiva de un conjunto.

```
lemma "¬(p2 - va (nm ''a'') → ?P)"
apply clarsimp
by (erule trans.cases, clarsimp+)+
```

Nótese cómo en este caso hemos utilizado una negación explícita. En la siguiente sección haremos la comparación con nuestra aproximación.

También podemos definir la sintaxis y la semántica de la lógica modal, como hemos hecho para CCS. Al efecto utilizamos un tipo de datos *formula* y un conjunto *sat* definido de forma inductiva que representa la relación de satisfacción.

```
datatype formula =  tt
                  | ff
                  | AndF formula formula ("_ ∧ _")
                  | OrF  formula formula ("_ ∨ _")
                  | Diam "action set" formula ("⟨ _ ⟩ _")
                  | Box  "action set" formula ("[ _ ] _")

inductive sat
 intros
  ttR[intro!]:   "P ⊨ tt"
  andR:  "⟦ P ⊨ Phi ; P ⊨ Psi ⟧ ⟹ P ⊨ (Phi ∧ Psi)"
  orR1:  "P ⊨ Phi ⟹  P ⊨ (Phi ∨ Psi)"
  orR2:  "P ⊨ Psi ⟹  P ⊨ (Phi ∨ Psi)"
  diamR: "⟦ P - A → Q ; A ∈ AS; Q ⊨ Phi ⟧ ⟹ P ⊨ ⟨ AS ⟩ Phi"
  boxR:  "(∀ Q A . ( (P - A → Q ∧ A ∈ AS) ⟶ Q ⊨ Phi)) ⟹ P ⊨ [ AS ] Phi"
```

Para probar que un proceso dado satisface una fórmula tenemos que realizar inducción sobre la relación *sat*. Esto se debe a la presencia del cuantificador universal en la regla *boxR*, el cual se refiere a todos los sucesores de un proceso. El método *ind_cases* aplica una instancia de la regla *cases* para el patrón proporcionado.

```
lemma "(va (nm ''a'') . va (nm ''b'') . 0)
       ⊨ [ {va (nm ''a'')} ]( ⟨{va (nm ''b'')}⟩ tt)"
apply (rule sat.boxR, clarify)
apply (ind_cases "A . P - B → Q")
by (force intro: trans.intros sat.intros)
```

A continuación se muestra cómo pueden demostrarse con Isabelle algunas de las propiedades de la máquina expendedora de la Sección 4.7.3.

```
defs Context_def[simp]:  "Context ≡
  And (Defi ''Ven'' (va (nm ''2p'') . Id ''VenB'' +
                     va (nm ''1p'') . Id ''VenL''))
 (And (Defi ''VenB'' (va (nm ''big'') . va (nm ''collectB'') .
```

```
                    Id ''Ven''))
 (And (Defi ''VenL'' (va (nm ''little'') . va (nm ''collectL'') .
                    Id ''Ven''))
  Empty))"

lemma "Id ''Ven'' ⊨ ⟨{va (nm ''2p'')}⟩ (⟨{va (nm ''big'')}⟩ tt)"
apply (force intro: trans.intros intro: sat.intros)
done

lemma "Id ''VenB'' ⊨ ( ⟨{va (nm ''big''), va (nm ''little'')} ⟩ tt)"
apply (force intro: trans.intros intro: sat.intros)
done

lemma "Id ''Ven'' ⊨ ( [ {va (nm ''big''),
                         va (nm ''little'')} ] ff)"
apply (rule sat.boxR, clarify)
by (erule trans.cases, simp_all,clarsimp)+

lemma "Id ''Ven'' ⊨ [{va (nm ''2p'')}]
      (([{va (nm ''little'')}] ff) ∧ (⟨{va (nm ''big'')}⟩ tt))"
apply (rule sat.boxR, clarify)
apply (erule trans.cases, simp_all,clarsimp)+
apply (rule sat.andR, rule sat.boxR, clarify)
apply (erule trans.cases, simp_all,clarsimp)+
apply (force intro: trans.intros intro: sat.intros)
apply (rule sat.andR, rule sat.boxR, clarify)
apply (erule trans.cases, simp_all,clarsimp)+
done

lemma "Id ''Ven'' ⊨ [{va (nm ''2p''),va (nm ''1p'')}]
       ( [{va (nm ''2p''),va (nm ''1p'')}] ff)"
apply (rule sat.boxR, clarify)
apply (ind_cases "Id X - A → Q", simp)
apply (ind_cases "P + Q - A → P'")
apply (ind_cases "A . P - B → Q", clarsimp)
apply (rule sat.boxR, clarify)
apply (ind_cases "Id X - A → Q", simp)
apply (ind_cases "A . P - B → Q", clarsimp)
apply (ind_cases "A . P - B → Q", clarsimp)
apply (rule sat.boxR, clarify)
apply (ind_cases "Id X - A → Q", simp)
apply (ind_cases "A . P - B → Q", clarsimp)
done
```

4.8.1. Comparación con la representación en Maude

Isabelle ha sido diseñado tanto como marco lógico como demostrador de teoremas. Por tanto, los conceptos de variables esquemáticas y unificación son básicos, de tal forma que el

usuario no tiene que preocuparse de ellos. Por el contrario, Maude no soporta directamente las variables desconocidas. Pero, como hemos visto, gracias a sus características reflexivas podemos representarlas en el lenguaje, para tratarlas creando nuevas metavariables, propagando sus valores cuando estos llegan a ser conocidos.

Isabelle/HOL ofrece además diversas herramientas automáticas que ayudan a la demostración de teoremas. Por ejemplo, como se ha visto anteriormente, Isabelle genera automáticamente una regla de inducción cuando se define de forma inductiva un conjunto. Obviamente, esto añade mucho poder expresivo. Pero el usuario tiene que saber cómo utilizar todas estas herramientas, es decir, cuándo ha de utilizar reglas de introducción o eliminación, o cuándo ha de utilizar inducción, por ejemplo. En Maude, solo utilizamos las reglas de reescritura que definen la semántica y la estrategia de búsqueda exhaustiva que (de una forma ciega) las utiliza, y ello nos permite probar sentencias sobre CCS y sobre la lógica modal, de una misma forma.

Como se comentó en la Sección 4.3.3, las demostraciones negativas (por ejemplo, que una transición CCS no es válida) están basadas en nuestro enfoque en la completitud de nuestra estrategia de búsqueda. En Isabelle un teorema con una negación explícita puede probarse utilizando inducción.

Isabelle, como marco lógico, utiliza una lógica de orden superior para metarrepresentar las lógicas del usuario. Es en esta metalógica donde la resolución tiene lugar. Gracias a las propiedades reflexivas de la lógica de reescritura, podemos bajar de este nivel superior, representando conceptos de order superior en un marco de primer orden. En cierto sentido, esta comparación puede resumirse diciendo que con nuestro ejemplo de CCS hemos mostrado cómo se pueden utilizar técnicas de order superior en un marco de primer orden, gracias a la reflexión; es decir, la reflexión aporta a un sistema de primer orden, como Maude, gran parte del poder expresivo de un sistema de orden superior, como Isabelle.

Más aún, la lógica de reescritura es una moneda de dos caras, donde la deducción también puede interpretarse como cómputo. Esto nos permite obtener información de las demostraciones, tal y como hicimos con la operación `succ` que devuelve los sucesores de un proceso. En cambio, aunque podemos representar en Isabelle el conjunto de sucesores de un proceso, y probar si un proceso dado pertenece a este conjunto, no podemos calcular (o mostrar) este conjunto.

Christine Röeckl también ha utilizado Isabelle/HOL en su tesis [Röe01b] para representar la semántica de CCS. Ella define la relación de transición de CCS de forma inductiva como lo hemos hecho nosotros, y define entonces relaciones de equivalencia entre procesos, como la bisimulación y la equivalencia observacional, y demuestra, utilizando Isabelle, propiedades sobre ellas. También ha utilizado Isabelle/HOL para representar el π-cálculo de diferentes maneras, con sintaxis abstractas de primer orden y de orden superior [Röe01b, Röe01a, RHB01].

El sistema similar HOL [GM93] ha sido utilizado por Monica Nesi para formalizar CCS con paso de valores [Nes99] y una lógica modal para este cálculo [Nes96]. En ambos casos, la autora está interesada en demostrar propiedades sobre las propias semánticas representadas. Melham también ha utilizado el demostrador de teoremas HOL para realizar una formalización del π-cálculo [Mel94].

4.9. Conclusiones

En este capítulo hemos mostrado con todo detalle cómo los sistemas de inferencia pueden representarse en lógica de reescritura y su lenguaje de implementación Maude de forma general y *completamente ejecutable* siguiendo el enfoque de *reglas de inferencia como reescrituras*. Para disponer de un caso de estudio concreto que nos permita ilustrar las ideas generales, hemos abordado la representación de la semántica operacional estructural de CCS. Hemos resuelto los problemas de la presencia de nuevas variables en la parte derecha de las reglas y el no determinismo por medio de las propiedades reflexivas de la lógica de reescritura y su realización en el módulo `META-LEVEL` de Maude. En particular, hemos utilizado metavariables, junto con la capacidad de sustitución de la operación de descenso `meta-apply`, para resolver el problema de las variables nuevas, y una estrategia de búsqueda que maneja árboles conceptuales de reescrituras para resolver el problema del no determinismo. Las soluciones presentadas no son dependientes de CCS y pueden utilizarse en particular para implementar una gran variedad de sistemas de inferencia y definiciones de semánticas operacionales.

Hemos visto también cómo la semántica puede ser extendida para responder preguntas sobre la capacidad de un proceso para realizar una acción, considerando metavariables como procesos, de la misma forma que habíamos hecho antes con las acciones. Tener metavariables como procesos nos permite extender la semántica a trazas, definir la semántica de transiciones débiles, y calcular cuáles son los sucesores de un proceso después de realizar una acción. Por otra parte, nos permite representar la semántica de la lógica modal de Hennessy-Milner de una forma muy similar a como se hace en su definición matemática.

Hemos comparado nuestro enfoque con el de Isabelle/HOL, llegando a la conclusión de que, aunque Maude, al ser un marco tan general no es quizás tan fácil de usar, sí que puede utilizarse para aplicar técnicas de orden superior en un marco de primer orden, por medio de la reflexión.

Cierto es que otras herramientas de próposito más específico, tales como la herramienta *Concurrency Workbench of the New Century (CWB-NC)* [CS02], son más eficientes y expresivas que nuestra herramienta, pues esta ha de verse como un prototipo en el que podemos "jugar" no solo con procesos y sus capacidades, sino con la semántica, representada a un nivel matemático muy alto, añadiendo o modificando ciertas reglas. El diseño de CWB-NC explota la independencia del lenguaje de sus rutinas de análisis localizando los procedimientos específicos de un lenguaje, lo cual permite a los usuarios introducir sus propios lenguajes de descripción de sistemas por medio de la herramienta *Process Algebra Compiler* [CMS95], que convierte la definición de la semántica operacional dada, en código SML. Nosotros hemos seguido un enfoque similar, aunque hemos intentado mantener la representación de la semántica a un nivel tan alto como ha sido posible, sin que por ello deje de ser ejecutable. De esta forma evitamos la necesidad de traducir la representación de la semántica a otros lenguajes ejecutables.

Capítulo 5

Implementación de CCS en Maude 2

En el Capítulo 4 hemos seguido el primero de los caminos posibles para implementar la semántica de CCS. Este capítulo describe en detalle cómo abordar la separación entre teoría y práctica en cuanto a ejecutabilidad de las especificaciones, en una nueva implementación de la semántica operacional de CCS en Maude, siguiendo el enfoque consistente en transformar la relación de transición entre estados del lenguaje en la relación de reescritura entre términos que representen estos estados. Así, la regla de inferencia

$$\frac{P_1 \to Q_1 \ldots P_n \to Q_n}{P_0 \to Q_0}$$

se representaría mediante la regla de reescritura condicional

$$P_0 \longrightarrow Q_0 \quad \textit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n,$$

donde la condición incluye reescrituras.

El que Maude 2.0 [CDE+00b] admita reglas condicionales con reescrituras en las condiciones que se resuelven en tiempo de ejecución por medio de un mecanismo de búsqueda predefinido (véase Sección 2.4), ha hecho posible que reconsideremos esta segunda posibilidad en la que las transiciones del lenguaje se convierten en reescrituras. Este capítulo detalla los resultados obtenidos siguiendo esta segunda alternativa y compara las dos implementaciones producidas. Veremos que la segunda implementación es algo más simple al ser más cercana a la presentación habitual matemática de las semánticas operacionales, como ya dijimos en el Capítulo 3.[1] Sin embargo, existe todavía la necesidad de salvar algunos huecos entre la teoría y la práctica, y en este sentido el nuevo atributo `frozen` disponible en Maude 2.0 ha jugado un papel muy importante, como se describe con detalle en la Sección 5.2.

[1] La extensión de Maude descrita en [Bra01] también utiliza reescrituras en las condiciones. Las ventajas obvias de Maude 2.0 son su generalidad y eficiencia. En lo que se refiere a nuestro trabajo en este capítulo, que sepamos el intérprete para MSOS no se ha utilizado hasta la fecha para ejecutar ninguna especificación de la semántica operacional de CCS.

El trabajo descrito en este capítulo se ha presentado en el *Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002*, publicándose en [VMO02b].

5.1. Sintaxis de CCS

Comenzamos con una segunda exposición de la sintaxis de CCS en Maude. La diferencia con la definición dada en el Capítulo 4 es que los operadores para construir procesos se definen ahora como *frozen*, una nueva característica de Maude 2.0. En la Sección 5.2 explicaremos la razón de esta declaración. Además, los operadores binarios `_+_` y `_|_` se han definido como asociativos y conmutativos, lo que reduce el número de reglas de la semántica.

```
fmod ACTION is
  protecting QID .
  sorts Label Act .
  subsorts Qid < Label < Act .
  op tau : -> Act .
  op ~_ : Label -> Label .
  eq ~ ~ L:Label = L:Label .
endfm

fmod PROCESS is
  protecting ACTION .
  sorts ProcessId Process .
  subsorts Qid < ProcessId < Process .
  op 0 : -> Process .
  op _._ : Act Process -> Process [frozen prec 25] .
  op _+_ : Process Process -> Process [frozen assoc comm prec 35] .
  op _|_ : Process Process -> Process [frozen assoc comm prec 30] .
  op _[_/_] : Process Label Label -> Process [frozen prec 20] .
  op _\_ : Process Label -> Process [frozen prec 20] .
endfm
```

Al igual que en el capítulo anterior, representamos CCS completo, incluyendo definiciones de procesos posiblemente recursivos a través de *contextos*. El siguiente módulo `CCS-CONTEXT`, en el que se definen estos contextos, junto con operaciones para manejarlos, es análogo al presentado en la Sección 4.2, solo que ahora utilizamos la sintaxis de Maude 2.0 que permite declarar al nivel de las familias las operaciones parciales, como `_&_`.

```
fmod CCS-CONTEXT is
  including PROCESS .

  sort Context .

  op _=def_ : ProcessId Process -> Context [prec 40] .
  op nil : -> Context .
  op _&_ : [Context] [Context] -> [Context] [assoc comm id: nil prec 42] .
```

```
  op _definedIn_ : ProcessId Context -> Bool .
  op def : ProcessId Context -> [Process] .
  op context : -> Context .

  vars X X' : ProcessId .
  var P : Process .
  vars C C' : Context .

  cmb (X =def P) & C : Context if not(X definedIn C) .

  eq X definedIn nil = false .
  eq X definedIn (X' =def P & C') = (X == X') or (X definedIn C') .
  eq def(X, (X' =def P) & C') = if X == X' then P else def(X, C') fi .
endfm
```

El módulo incluye una constante `context`, utilizada para guardar la definición de los identificadores de proceso en cada especificación CCS.

5.2. Semántica de CCS

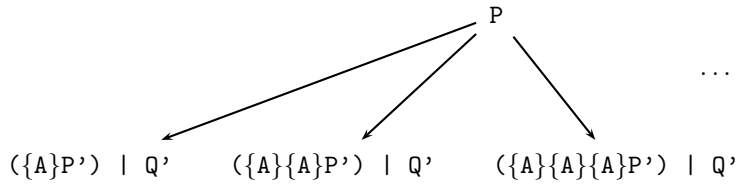
Para implementar la semántica de CCS tomando las transiciones como reescrituras, interpretaremos una transición $P \xrightarrow{a} P'$ como una reescritura en lógica de reescritura. Sin embargo, las reescrituras no contienen etiquetas, las cuales son esenciales en la semántica de CCS. Por tanto, haremos que la etiqueta sea parte del término resultado, obteniendo de esta forma reescrituras de la forma $P \longrightarrow \{a\}P'$, donde $\{a\}P'$ es un valor del tipo `ActProcess`, un supertipo de `Process`, lo que explicaremos más adelante. El siguiente módulo, que es admisible [CDE+00b] y por tanto directamente ejecutable, incluye la implementación de la semántica de CCS:

```
mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .

  sort ActProcess .
  subsort Process < ActProcess .

  op {_}_ : Act ActProcess -> ActProcess [frozen] .

  vars L M : Label .
  var A : Act .
  vars P P' Q Q' R : Process .
  var X : ProcessId .

  *** Prefix
  rl [pref] : A . P => {A}P .

  *** Summation
  crl [sum] : P + Q => {A}P'
           if P => {A}P' .
```

```
  *** Composition
  crl [par] : P | Q => {A}(P' | Q)
          if P => {A}P' .
  crl [par] : P | Q => {tau}(P' | Q')
          if P => {L}P' /\ Q => {~ L}Q' .

  *** Relabelling
  crl [rel] : P[M / L] => {M}(P'[M / L])
          if P =>{L}P' .
  crl [rel] : P[M / L] => {~ M}(P'[M / L])
          if P =>{~ L}P' .
  crl [rel] : P[M / L] => {A}(P'[M / L])
          if P =>{A}P' /\ A =/= L /\ A =/= ~ L .

  *** Restriction
  crl [res] : P \ L => {A}(P' \ L) if P => {A}P' /\ A =/= L /\ A =/= ~ L .

  *** Definition
  crl [def] : X => {A}P if (X definedIn context) /\ def(X,context) => {A}P .
endm
```

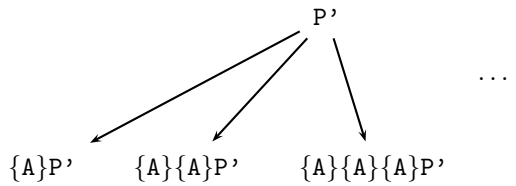
En esta representación de la semántica, las reglas de reescritura tiene la propiedad de "aumentar el tipo", es decir, en una reescritura $t \longrightarrow t'$, el menor tipo del término t' es mayor (en la relación de subtipado) que el menor tipo de t. Por tanto, una regla no puede ser aplicada a menos que el término resultante esté bien formado. Por ejemplo, aunque `A . P` $\longrightarrow$ `{A}P` es una reescritura correcta, no podemos derivar `(A . P) | Q` $\longrightarrow$ `({A}P) | Q`, ya que el término de la parte derecha no está bien formado. En consecuencia las reescrituras solo pueden ocurrir al nivel más externo de un término que representa un proceso, y no dentro del término.

Pero este mecanismo de "aumentar el tipo" no es suficiente en la versión del sistema Maude 2.0 si tenemos reescrituras en las condiciones y procesos infinitos. Si nos encontramos con la condición de reescritura `P => {A}Q`, el sistema intentará reescribir `P` de cada forma posible, y el resultado se intentará encajar con el patrón `{A}Q`. Si en una aplicación concreta de esta regla `P` fuese `(A . P') | Q'`, entonces el término sería reescrito a `({A}P') | Q'` aunque después el resultado se rechazaría al no encajar con el patrón. El problema aparecería cuando nos encontraramos con procesos recursivos, para los que la búsqueda predefinida que intenta satisfacer la condición de reescritura puede convertirse en una búsqueda infinita, que no terminaría nunca. Por ejemplo, si el proceso `P'` anterior viene dado por `P' = A . P'`, entonces `P` se reescribirá a `({A}P') | Q'`, `({A}{A}P') | Q'`, etc.,



aunque todos los resultados terminen siendo rechazados, por no estar bien formados. Nuestra solución a este problema ha sido declarar todos los operadores de la sintaxis como operadores `frozen`, lo cual evita que los argumentos de los correspondientes operadores sean reescritos por reglas.

Sin embargo, el problema sigue apareciendo cuando queremos conocer *todas* las posibles reescrituras del proceso `P'` definido anteriormente, que sean de la forma `{A}Q` (con `Q` de tipo `Process`). En tal caso, el proceso `P'` se reescribe a `{A}P'`, pero también a `{A}{A}P'`, `{A}{A}{A}P'`, etc.



y solo la primera reescritura encaja con el patrón `{A}Q` . En consecuencia, también tenemos que declarar como `frozen` el operador `{_}_`.

En resumen, utilizamos el atributo `frozen` para evitar un bucle infinito en el proceso de búsqueda cuando sabemos que la misma no tendría éxito, lo que puede suceder por dos razones diferentes: bien porque los términos construidos no estén bien formados, como en `({A}P') | Q'`, y esa es la razón por la cual declaramos los operadores de la sintaxis como `frozen`; o porque los términos no encajen con el patrón dado, como en `{A}{A}P'`, y esa es la razón por la cual se declara como `frozen` el operador `{_}_`.

Desgraciadamente, al declarar todos los operadores constructores como `frozen`, hemos perdido la posibilidad de probar que un proceso puede realizar una secuencia de acciones, o *traza*, pues las reglas solo pueden utilizarse para obtener sucesores *en un paso*. La regla de congruencia de la lógica de reescritura no puede utilizarse al ser los operadores `frozen` y la de transitividad no puede utilizarse, porque todas las reglas reescriben a algo de la forma `{A}Q`, y no hay ninguna regla con este patrón en la parte izquierda. Afortunadamene, esto no es problema si queremos utilizar la semántica solo en la definición de la semántica de la lógica modal, pues allí únicamente se necesitan los sucesores en un paso.

Sin embargo, también podemos solventar el problema extendiendo la semántica con reglas que generen el cierre transitivo de las transiciones CCS, de la siguiente manera:

```
sort TProcess .
subsort TProcess < ActProcess .

op [_] : Process -> TProcess [frozen] .

crl [refl] : [ P ] => {A}Q  if P => {A}Q .
crl [tran] : [ P ] => {A}AP if P => {A}Q /\ [ Q ] => AP .
```

Nótese la utilización del operador *ficticio* `[_]` para indicar qué reglas queremos que se utilicen para resolver una condición. Si no lo utilizáramos en la parte izquierda de las reglas anteriores, la parte izquierda de la cabeza de las reglas y de las reescrituras en las

condiciones serían variables que encajarían con cualquier término y entonces la misma regla podría ser utilizada para resolver su primera condición, dando lugar a un bucle infinito. Además, el operador ficticio también ha sido declarado como `frozen` para evitar reescrituras sin sentido como por ejemplo `[ A . P ]` $\longrightarrow$ `[ {A}P ]`.

La representación que hemos obtenido de CCS, incluyendo estas dos últimas reglas, es semánticamente correcta, en el sentido de que dado un proceso CCS P, existirán procesos $P_1, \ldots, P_k$ tales que

$$P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} P_k$$

si y solo si `[ P ]` puede ser reescrito a `{a1}{a2}...{ak}Pk` (véase [MOM93]).

Utilizando el comando `search` de Maude 2.0, podemos encontrar todos los posibles sucesores en un paso de un proceso, o todos los sucesores tras realizar una acción dada.

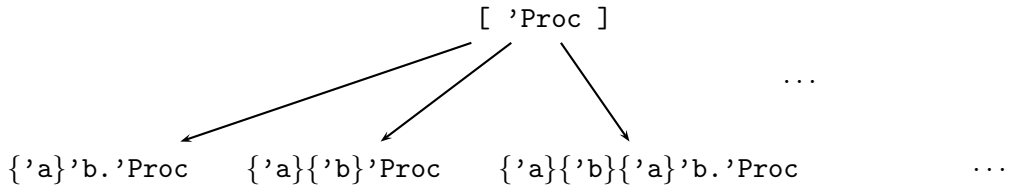
```
Maude> search 'a . 'b . 0 | ~ 'a . 0 => AP:ActProcess .
Solution 1 (state 1)
  AP:ActProcess --> {~ 'a}0 | 'a . 'b . 0

Solution 2 (state 2)
  AP:ActProcess --> {'a}'b . 0 | ~ 'a . 0

Solution 3 (state 3)
  AP:ActProcess --> {tau}0 | 'b . 0

No more solutions.

Maude> search 'a . 'b . 0 + 'c . 0 => {'a}AP:ActProcess .

Solution 1 (state 2)
  AP:ActProcess --> 'b . 0

No more solutions.
```

Si añadimos la siguiente ecuación al módulo `CCS-SEMANTICS`, que define el proceso recursivo `'Proc` en el contexto de CCS, podemos probar que `'Proc` puede realizar la traza `'a 'b 'a`:

```
  eq context =  'Proc =def 'a . 'b . 'Proc .

Maude> search [1] [ 'Proc ] => {'a}{'b}{'a}X:Process .

Solution 1 (state 5)
  X:Process --> 'b . 'Proc
```

En concreto, por medio de este último comando hemos pedido al sistema Maude que busque la primera forma, `[1]`, en la cual el término `[ 'Proc ]` puede ser reescrito al patrón `{'a}{'b}{'a}X:Process`. El comando `search` realiza una búsqueda *en anchura* en el árbol conceptual de todas las posibles reescrituras del término `[ 'Proc ]`, y ya que existe una solución, la encuentra. Sin embargo, si pidiéramos que buscara más soluciones

de esta misma consulta, la búsqueda no terminaría a pesar de que no las hay, ya que en su busca se exploraría el árbol completo de búsqueda que es infinito.



5.3. Extensión a la semántica de transiciones débiles

Consideraremos ahora la relación de transición $P \stackrel{a}{\Longrightarrow} P'$, definida en la Figura 4.3. También podemos implementar esta relación por medio de reescrituras, con lo que una transición $P \stackrel{a}{\longrightarrow}{}^{*} P$ se representará como una reescritura $P \longrightarrow \{a\}* P'$ y una transición $P \stackrel{a}{\Longrightarrow} P'$ se representará como una reescritura $P \longrightarrow \{\{a\}\}P'$. De nuevo tenemos que introducir operadores ficticios para prevenir la utilización de las nuevas reglas de reescritura de manera incorrecta en la verificación de las condiciones de reescritura. La implementación propuesta es la siguiente:

```
sorts Act*Process ObsActProcess .

op {_}*_  : Act Process -> Act*Process [frozen] .
op {{_}}_ : Act Process -> ObsActProcess [frozen] .

sort WProcess .
subsorts WProcess < Act*Process ObsActProcess .

op |_| : Process -> WProcess [frozen] .
op <_> : Process -> WProcess [frozen] .

rl  [refl*] : | P | => {tau}* P .
crl [tran*] : | P | => {tau}* R
           if   P   => {tau}Q /\
              | Q | => {tau}* R .

crl [weak] : < P > => {{A}}P'
          if | P |  => {tau}* Q  /\
               Q    => {A}Q'     /\
             | Q' | => {tau}* P' .
```

Obsérvese que tanto los nuevos operadores semánticos, `{_}*_` y `{{_}}_`, como los operadores ficticios, `|_|` y `<_>`, se declaran también como `frozen`, por las mismas razones ya explicadas en la Sección 5.2.

Podemos utilizar el comando `search` para buscar todos los sucesores débiles de un proceso dado después de realizar una acción `'a`.

```
Maude> search < tau . 'a . tau . 'b . 0 > => {{ 'a }}AP:ActProcess .

Solution 1 (state 2)
  AP:ActProcess --> tau . 'b . 0

Solution 2 (state 3)
  AP:ActProcess --> 'b . 0

No more solutions.
```

5.4. Lógica modal de Hennessy-Milner

Ahora queremos implementar la lógica modal de Hennessy-Milner para describir capacidades locales de procesos CCS [Sti96]. Las fórmulas son construidas de la siguiente manera:

$$\Phi ::= \mathtt{tt} \mid \mathtt{ff} \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid \langle K\rangle\Phi \mid [\![K]\!]\Phi \mid \langle\langle K\rangle\rangle\Phi$$

donde K es un conjunto de acciones. La relación de satisfacción que describe cuando un proceso P satisface una propiedad Φ, $P \models \Phi$, se ha definido de forma inductiva en las Figuras 4.4 y 4.6. En la Sección 4.7 se explicó el significado de todas las fórmulas.

Ya que la definición de la relación de satisfacción utiliza las transiciones de CCS, podríamos intentar implementarla al mismo nivel. Por ejemplo, las dos definiciones siguientes

$$\begin{array}{ll} P \models \Phi_1 \wedge \Phi_2 & \text{iff } P \models \Phi_1 \text{ and } P \models \Phi_2 \\ P \models \langle K\rangle\Phi & \text{iff } \exists Q \in \{P' \mid P \xrightarrow{a} P' \wedge a \in K\}\,.\, Q \models \Phi \end{array}$$

podrían implementarse con las siguientes reglas:

```
rl [and] : P |= Phi1 /\ Phi2 => true if P |= Phi1 => true /\
                                        P |= Phi2 => true .
rl [dia] : P |= < A > Phi => true if P => {A}Q /\ Q |= Phi => true .
```

Estas reglas son correctas y representan exactamente lo que expresa la relación de satisfacción de la lógica modal. Por ejemplo, la condición de la segunda regla representa que *existe* un proceso Q tal que $P \xrightarrow{a} Q$ y $Q \models \Phi$, que es la definición del operador modal diamante. Esto es así porque la variable `Q` está cuantificada existencialmente en la condición de la regla. Pero encontramos un problema con la definición del operador modal universal $[K]$, al aparecer un cuantificador universal sobre todas las posibles transiciones de un proceso. Si queremos trabajar con todas las posibles reescrituras en un paso de un término, tenemos que subir al metanivel. Utilizando la operación `metaSearch`, hemos definido una operación `succ` que devuelve todos los sucesores (metarrepresentados) de un proceso tras realizar acciones dentro de un conjunto de acciones dado.

La evaluación de `allOneStep(T, N, X)` devuelve todas las reescrituras en un paso del término `T` (saltándose las primeras `N` soluciones) que encajan con el patrón `X` utilizando

las reglas de reescritura del módulo `MOD` (la metarrepresentación de `CCS-SEMANTICS`, denotada por el término `['CCS-SEMANTICS]` en el módulo `SUCC` expuesto a continuación). La evaluación de `filter(F, TS, AS)` devuelve los procesos metarrepresentados `P` tales que `F[A, P]` está en `TS` y `A` está en `AS`. A la hora de buscar el término `A` en el conjunto de términos `AS`, los términos se comparan en el módulo `MOD`. Esto es debido a que los términos metarrepresentados diferentes `''a.Qid` y `''a.Act` representan la misma acción dentro del módulo `CCS-SEMANTICS`. La operación `filter` se utiliza en la definición de `succ(T,TS)` para eliminar del conjunto de sucesores del proceso `T` aquellos procesos que son alcanzados realizando una acción que no esté en el conjunto `TS`.

Habiendo definido estas operaciones de una forma tan general, también podemos implementar una operación `wsucc` que devuelva todos los sucesores débiles de un proceso.

```
fmod SUCC is
  including META-LEVEL .

  op MOD : -> Module .
  eq MOD = ['CCS-SEMANTICS] .

  sort TermSet .
  subsort Term < TermSet .

  op mt : -> TermSet .
  op _+_ : TermSet TermSet -> TermSet [assoc comm id: mt] .
  op _isIn_ : Term TermSet -> Bool .
  op allOneStep : Term Nat Term -> TermSet .
  op filter : Qid TermSet TermSet -> TermSet .
  op succ : Term TermSet -> TermSet .
  op wsucc : Term TermSet -> TermSet .

  var M : Module .
  var F : Qid .
  vars T T' X : Term .
  var N : Nat .
  vars TS AS : TermSet .

  eq T isIn mt = false .
  eq T isIn (T' + TS) =
     (getTerm(metaReduce(MOD, '_==_[T,T'])) == 'true.Bool) or (T isIn TS) .

  eq filter(F, mt, AS) = mt .
  ceq filter(F, X + TS, AS) =
      (if T isIn AS then  T' else mt fi) + filter(F,TS,AS) if F[T,T'] := X .

  eq allOneStep(T,N,X) =
     if metaSearch(MOD,T, X, nil, '+,1,N) == failure then mt
     else  getTerm(metaSearch(MOD,T, X, nil, '+,1,N)) +
           allOneStep(T,N + 1,X)
     fi .
```

```
  eq succ(T,TS) = filter((''{_'}_), allOneStep(T,0,'AP:ActProcess),TS) .
  eq wsucc(T,TS) = filter((''{'{_'}'}_),
                          allOneStep('<_>[T],0,'OAP:ObsActProcess),TS) .
endfm
```

Utilizando las operaciones `succ` y `wsucc` hemos definido ecuacionalmente la relación de satisfacción de la lógica modal. Obsérvese cómo la semántica de los operadores modales se ha definido desplegando una conjunción o disyunción, que maneja los sucesores o sucesores débiles del proceso dado.

```
fmod MODAL-LOGIC is
  protecting SUCC .

  sort HMFormula .

  ops tt ff : -> HMFormula .
  ops _/\_  _\/_ : HMFormula HMFormula -> HMFormula .
  ops <_>_   '[_']_     : TermSet HMFormula -> HMFormula .
  ops <<_>>_ '['[_']']_ : TermSet HMFormula -> HMFormula .
  ops forall exists : TermSet HMFormula -> Bool .
  op _|=_ : Term HMFormula -> Bool .

  var P : Term .
  var K PS : TermSet .
  vars Phi Psi : HMFormula .

  eq P |= tt          = true .
  eq P |= ff          = false .
  eq P |= Phi /\ Psi  = P |= Phi and P |= Psi .
  eq P |= Phi \/ Psi  = P |= Phi or  P |= Psi .
  eq P |= [ K ] Phi   = forall(succ(P, K), Phi) .
  eq P |= < K > Phi   = exists(succ(P, K), Phi) .
  eq P |= [[ K ]] Phi = forall(wsucc(P, K), Phi) .
  eq P |= << K >> Phi = exists(wsucc(P, K), Phi) .

  eq forall(mt, Phi)  = true .
  eq forall(P + PS, Phi) = P |= Phi and forall(PS, Phi) .

  eq exists(mt, Phi)  = false .
  eq exists(P + PS, Phi) = P |= Phi or  exists(PS,Phi) .
endfm
```

Utilizando los mismos ejemplos de [Sti96] ya presentados en las Secciones 4.7.3 y 4.7.4, podemos ilustrar cómo se prueba en Maude que un proceso CCS satisface una fórmula modal. El primer ejemplo trata sobre una máquina expendedora `'Ven`, definida en un contexto CCS en la página 97.

La máquina satisface que tras insertar una moneda y pulsar un botón, se puede recoger un producto (grande o pequeño).

```
Maude> red ''Ven.Qid |= [ ''1p.Act + ''2p.Act ]
                        [ ''big.Act + ''little.Act ]
                        < ''collectB.Act + ''collectL.Act > tt .
result Bool: true
```

El segundo ejemplo trata el control de un cruce de una vía férrea y una carretera, y se especificó en la página 99. El proceso `'Crossing` satisface que cuando han llegado al cruce a la vez un coche y un tren, exactamente uno de ellos tiene posibilidad de cruzarlo.

```
Maude> red ''Crossing.Qid |= [[ ''car.Act ]] [[ ''train.Act ]]
           ((<< '~_[''ccross.Act] >> tt) \/ (<< '~_[''tcross.Act] >> tt)) .
result Bool: true

Maude> red ''Crossing.Qid |= [[ ''car.Act ]] [[ ''train.Act ]]
           ((<< '~_[''ccross.Act] >> tt) /\ (<< '~_[''tcross.Act] >> tt)) .
result Bool: false
```

Maude 2.0 tarda ahora 1,3 segundos en resolver el último comando. Con la primera implementación de CCS en Maude presentada en el Capítulo 4, se tardaba 10 minutos. La mejora es considerable, y es debida sobre todo al cambio de representación, si bien hay que tener en cuenta que la implementación de Maude 2.0 también intenta ser más eficiente en general. En la siguiente sección comparamos ambas implementaciones.

5.5. Conclusiones

Hemos presentado ya dos implementaciones diferentes de la semántica de CCS y su utilización para implementar la lógica modal de Hennessy-Milner; en el Capítulo 4 seguimos el enfoque de reglas de inferencia como reescrituras, mientras que en este hemos seguido el enfoque de transiciones como reescrituras.

En nuestra opinión la segunda implementación conlleva varias ventajas. En primer lugar es más cercana a la presentación matemática de la semántica. Una regla de la semántica operacional establece que la transición en la conclusión es posible si las transiciones en las premisas son posibles, y esa es precisamente la interpretación de una regla de reescritura condicional con reescrituras en las condiciones.

El primer enfoque necesita estructuras auxiliares como los multiconjuntos de juicios que tienen que ser probados como válidos, y mecanismos como la generación de nuevas metavariables y su propagación cuando sus valores concretos llegan a ser conocidos. Esto nos obligó a implementar al metanivel una estrategia de búsqueda que compruebe si un multiconjunto de juicios dado puede ser reducido al conjunto vacío de juicios, generando nuevas metavariables cada vez que se necesiten (véase Sección 4.3.2). La necesidad de introducir las metavariables nuevas es lo que hace que no podamos prescindir de la estrategia no estándar definida por nosotros.

Incluso si utilizáramos el sistema Maude 2.0 no podríamos usar su comando `search` pues este no puede manejar reglas de reescritura con nuevas variables en la parte derecha,

si estas no se vinculan a un valor en la resolución de alguna de sus condiciones, y eso es exactamente lo que ocurría.

En la segunda implementación la necesidad de la búsqueda aparece en las condiciones de reescritura, pero ahora el sistema Maude 2.0, que sí es capaz de trabajar con estas condiciones de reescritura junto con la presencia de variables nuevas ligadas en alguna condición, sí que resuelve el problema.

También existen diferencias en lo que sucede al nivel objeto (nivel de la representación de la semántica) y al metanivel (utilizando reflexión). En la primera implementación, la estrategia de búsqueda recorre el árbol conceptual que contiene todas las posibles reescrituras de un término, moviéndose continuamente entre el nivel objeto y el metanivel. En la implementación descrita en este capítulo, la búsqueda se realiza moviéndonos siempre en el nivel objeto, lo que la hace más rápida y simple.

La generalidad de las técnicas desarrolladas quedará de manifiesto en los capítulos siguientes.

Capítulo 6

Otros lenguajes de programación

En este capítulo vamos a presentar implementaciones de distintas semánticas operacionales estructurales, que van desde semánticas de evaluación a semánticas de computación con máquina abstracta, descritas en [Hen90], tanto para lenguajes funcionales como imperativos, incluyendo no determinismo. También presentamos una implementación de la semántica de evaluación del lenguaje Mini-ML descrito en [Kah87].

Todas las implementaciones que vamos a presentar en este capítulo siguen el enfoque de transiciones como reescrituras que, como ya hemos comentado, presenta diversas ventajas sobre el enfoque de reglas de inferencia como reescrituras, tanto a nivel metodológico como de eficiencia en la implementación.

6.1. El lenguaje funcional *Fpl*

El lenguaje *Fpl* (abreviatura de *Functional programming language*) es una extensión del lenguaje de expresiones *Exp4*, utilizado como ejemplo en el Capítulo 3, con declaración de funciones definidas por el usuario (con posibilidad de recursión mútua) y llamadas a funciones.

Su sintaxis abstracta se describe en la Figura 6.1. Un *programa* está formado por una expresión junto con una declaración, $\langle e, D\rangle$. De forma intuitiva, D proporciona las definiciones de todos los nombres de función en e.[1]

La sintaxis se implementa en el siguiente módulo funcional `FPL-SYNTAX`. Nótese que la estructura de la signatura se corresponde con la estructura de la gramática que define la sintaxis del lenguaje (Figura 6.1).

```
fmod FPL-SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp FunVar
```

[1] Utilizando axiomas de pertenencia podríamos definir cuándo un programa $\langle e, D\rangle$ es correcto, lo que sucedería cuando todas las funciones utilizadas en e estén definidas en D, de forma similar a como hemos definido los contextos correctos para CCS (Sección 4.2).

1. Categorías sintácticas

p	$\in$	*Prog*	op	$\in$	*Op*
D	$\in$	*Dec*	bop	$\in$	*BOp*
e	$\in$	*Exp*	n	$\in$	*Num*
be	$\in$	*BExp*	x	$\in$	*Var*
F	$\in$	*FunVar*	bx	$\in$	*BVar*

2. Definiciones

$$
\begin{array}{rcl}
p & ::= & \langle e, D \rangle \\
D & ::= & F(x_1, \ldots, x_k) \Longleftarrow e \mid F(x_1, \ldots, x_k) \Longleftarrow e, D \\
op & ::= & + \mid - \mid * \\
bop & ::= & And \mid Or \\
e & ::= & n \mid x \mid e' \; op \; e'' \mid \mathsf{If}\ be\ \mathsf{Then}\ e'\ \mathsf{Else}\ e'' \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e'' \mid F(e_1, \ldots, e_k) \\
be & ::= & bx \mid \mathsf{T} \mid \mathsf{F} \mid be' \; bop \; be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e, e')
\end{array}
$$

Figura 6.1: Sintaxis abstracta de *Fpl*.

```
      VarList NumList ExpList Prog Dec .

op V : Qid -> Var .
subsort Var < Exp .
subsort Num < Exp .

op BV : Qid -> BVar .
subsort BVar < BExp .
subsort Boolean < BExp .

op FV : Qid -> FunVar .

ops + - * : -> Op .

op 0 : -> Num .
op s : Num -> Num .

subsort Exp < ExpList .
op _,_ : ExpList ExpList -> ExpList [assoc prec 30] .

op ___ : Exp Op Exp -> Exp [prec 20] .
op If_Then_Else_ : BExp Exp Exp -> Exp [prec 25] .
op let_=_in_ : Var Exp Exp -> Exp [prec 25] .
op _`(_`) : FunVar ExpList -> Exp [prec 15] .

ops T F : -> Boolean .
ops And Or : -> BOp .
op ___ : BExp BOp BExp -> BExp [prec 20] .
op Not_ : BExp -> BExp [prec 15] .
op Equal : Exp Exp -> BExp .

subsort Var < VarList .
```

```
  op _,_ : VarList VarList -> VarList [assoc prec 30] .
  subsort VarList < ExpList .

  subsort Num < NumList .
  op _,_ : NumList NumList -> NumList [assoc prec 30] .
  subsort NumList < ExpList .

  op <_,_> : Exp Dec -> Prog .
  op nil : -> Dec .
  op _`(_`)<=_ : FunVar VarList Exp -> Dec [prec 30] .
  op _&_ : Dec Dec -> Dec [assoc comm id: nil prec 40] .

  op exDec1 : -> Dec .
  eq exDec1 =
    FV('Fac)(V('x)) <= If Equal(V('x),0) Then s(0)
                       Else V('x) * FV('Fac)(V('x) - s(0)) &
    FV('Rem)(V('x) , V('y)) <= If Equal(V('x),V('y)) Then 0
                               Else If Equal(V('y) - V('x), 0) Then V('y)
                                    Else FV('Rem)(V('x) , V('y) - V('x)) &
    FV('Double)(V('x)) <= V('x) + V('x) .
endfm
```

Además de la sintaxis completa de *Fpl*, el módulo anterior incluye una constante `exDec1`, con un ejemplo de conjunto de declaraciones de función que utilizaremos más tarde.

La sintaxis abstracta de *Fpl* de la Figura 6.1 es común para las tres definiciones semánticas presentadas en [Hen90], y que vamos a ver a continuación. Sin embargo, para facilitar la representación en Maude 2.0 de las diferentes semánticas habrá que hacer algunos cambios en el módulo `FPL-SYNTAX`. Estos cambios podrían haberse hecho desde el principio, pero preferimos tener distintas versiones para hacer notar las (pequeñas) diferencias.

Las diferentes semánticas para el lenguaje *Fpl* que vamos a ver son las siguientes: una semántica de evaluación (o paso largo) bastante abstracta, una semántica de computación (o paso corto) más detallada, y una tercera semántica aún más concreta en forma de una *máquina abstracta*.

6.1.1. Semántica de evaluación

Para poder evaluar una expresión numérica e es necesario contar con un entorno ρ que asigne valor a las variables en la expresión e, y con una declaración que proporcione un contexto para los símbolos de función en e. Por tanto, los juicios de esta primera semántica serán de la forma

$$D, \rho \vdash e \Longrightarrow_A v.$$

Lo mismo ocurre con las expresiones booleanas ya que, aunque las llamadas a función son solo expresiones numéricas, las expresiones numéricas se utilizan también para construir

$$\text{CR}\quad \frac{}{D,\rho \vdash \mathbf{n} \Longrightarrow_A \mathbf{n}}$$

$$\text{VarR}\quad \frac{}{D,\rho \vdash x \Longrightarrow_A \rho(x)}$$

$$\text{OpR}\quad \frac{D,\rho \vdash e \Longrightarrow_A v \qquad D,\rho \vdash e' \Longrightarrow_A v'}{D,\rho \vdash e\ op\ e' \Longrightarrow_A Ap(op,v,v')}$$

$$\text{IfR}\quad \frac{D,\rho \vdash be \Longrightarrow_B \mathsf{T} \qquad D,\rho \vdash e \Longrightarrow_A v}{D,\rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \Longrightarrow_A v}$$

$$\frac{D,\rho \vdash be \Longrightarrow_B \mathsf{F} \qquad D,\rho \vdash e' \Longrightarrow_A v'}{D,\rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \Longrightarrow_A v'}$$

$$\text{LocR}\quad \frac{D,\rho \vdash e \Longrightarrow_A v \qquad D,\rho[v/x] \vdash e' \Longrightarrow_A v'}{D,\rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \Longrightarrow_A v'}$$

$$\text{FunR}\quad \frac{D,\rho \vdash e_i \Longrightarrow_A v_i,\ 1 \le i \le k \qquad D,\rho[v_1/x_1,\ldots,v_k/x_k] \vdash e \Longrightarrow_A v}{D,\rho \vdash F(e_1,\ldots,e_k) \Longrightarrow_A v}\ F(x_1,\ldots,x_k) \Leftarrow e \text{ está en } D$$

Figura 6.2: Semántica de evaluación para *Fpl*, $\Longrightarrow_A$.

expresiones booleanas con el operador Equal. Los juicios para evaluar expresiones booleanas serán de la forma

$$D,\rho \vdash be \Longrightarrow_B bv.$$

Tendremos que $\rho \vdash \langle e, D\rangle \Longrightarrow v$ si y solo si $D,\rho \vdash e \Longrightarrow_A v$. Las reglas semánticas para la transición $\Longrightarrow_A$ se muestran en la Figura 6.2, y las correspondientes a la transición $\Longrightarrow_B$ en la Figura 6.3.

La regla FunR indica que para evaluar $F(e_1,\ldots,e_k)$ primero hay que evaluar todos los argumentos y después evaluar el cuerpo de la definición de F, en un entorno en el que los parámetros formales se han ligado al valor del correspondiente parámetro actual. Este es el mecanismo de paso de parámetros conocido como *llamada por valor*. Más tarde veremos otra alternativa denominada *llamada por nombre*.

Esta regla presenta un problema a la hora de implementarla. El mismo radica en que el número de premisas no es fijo, dependiendo de la llamada a función concreta que queramos evaluar, concretamente del número de argumentos que esta tenga. En Maude 2.0 no podemos escribir una regla de reescritura condicional con un número no determinado de condiciones, a no ser que lo hicieramos utilizando una función definida al metanivel que construyera las correspondientes reglas de reescritura cuando el número de condiciones fuese concretado.

Sin embargo, también podemos resolver el problema si vemos la lista de parámetros actuales como una nueva categoría sintáctica de listas no vacías de expresiones numéricas, y

$$\text{BCR} \quad \frac{}{D, \rho \vdash \mathsf{T} \Longrightarrow_B \mathsf{T}} \qquad \frac{}{D, \rho \vdash \mathsf{F} \Longrightarrow_B \mathsf{F}}$$

$$\text{BVarR} \quad \frac{}{D, \rho \vdash bx \Longrightarrow_B \rho(bx)}$$

$$\text{BOpR} \quad \frac{D, \rho \vdash be \Longrightarrow_B bv \quad D, \rho \vdash be' \Longrightarrow_B bv'}{D, \rho \vdash be\ bop\ be' \Longrightarrow_B Ap(bop, bv, bv')}$$

$$\text{NotR} \quad \frac{D, \rho \vdash be \Longrightarrow_B \mathsf{T}}{D, \rho \vdash \mathsf{Not}\ be \Longrightarrow_B \mathsf{F}} \qquad \frac{D, \rho \vdash be \Longrightarrow_B \mathsf{F}}{D, \rho \vdash \mathsf{Not}\ be \Longrightarrow_B \mathsf{T}}$$

$$\text{EqR} \quad \frac{D, \rho \vdash e \Longrightarrow_A v \quad D, \rho \vdash e' \Longrightarrow_A v}{D, \rho \vdash \mathsf{Equal}(e, e') \Longrightarrow_B \mathsf{T}} \qquad \frac{D, \rho \vdash e \Longrightarrow_A v \quad D, \rho \vdash e' \Longrightarrow_A v'}{D, \rho \vdash \mathsf{Equal}(e, e') \Longrightarrow_B \mathsf{F}}\ v \neq v'$$

Figura 6.3: Semántica de evaluación para expresiones booleanas, $\Longrightarrow_B$.

escribimos una regla semántica que evalúe listas de expresiones. La regla FunR modificada y la regla ExpLR para evaluar listas de expresiones son las siguientes:

$$\text{FunR} \quad \frac{D, \rho \vdash el \Longrightarrow_A vl \quad D, \rho[vl/xl] \vdash e \Longrightarrow_A v}{D, \rho \vdash F(el) \Longrightarrow_A v}\ F(xl) \Leftarrow e \text{ está en } D$$

$$\text{ExpLR} \quad \frac{D, \rho \vdash e \Longrightarrow_A v \qquad D, \rho \vdash el \Longrightarrow_A vl}{D, \rho \vdash e, el \Longrightarrow_A v, vl}$$

La semántica utiliza una función *Ap* para aplicar un operador binario a dos argumentos, definida como en el módulo `AP` de la Sección 3.2.

Los entornos de variables y la función de modificación de un entorno añadiendo ligaduras o modificando ligaduras anteriores también se implementaron en el módulo `ENV` de la Sección 3.2. En este caso habría que extender la operación de modificación para permitir la inclusión de una lista de nuevas ligaduras.

El siguiente módulo `EVALUATION` contiene las reglas de reescritura que representan la semántica de evaluación de *Fpl*, tanto para expresiones numéricas como booleanas.

```
mod EVALUATION is
  protecting AP .
  protecting ENV .

  sort Statement .
  subsorts Num Boolean NumList < Statement .
  op _,_|-_ : Dec ENV Exp -> Statement [prec 40] .
  op _,_|-_ : Dec ENV BExp -> Statement [prec 40] .
  op _,_|-_ : Dec ENV ExpList -> Statement [prec 40] .
```

```
vars D D' : Dec .
var ro : ENV .
var n : Num .
var x  : Var .
var bx : BVar .
var v v' : Num .
var bv bv' : Boolean .
var op : Op .
vars e e' : Exp .
vars be be' : BExp .
var bop : BOp .
var F : FunVar .
var el : ExpList .
var xl : VarList .
var vl : NumList .

*** Evaluation semantics for Fpl

rl [CR] : D,ro |- n => n .

rl [VarR] : D,ro |- x => ro(x) .

crl [OpR] : D,ro |- e op e' => Ap(op,v,v')
         if D,ro |- e  => v  /\
            D,ro |- e' => v' .

crl [IfR1] : D,ro |- If be Then e Else e' => v
          if D,ro |- be => T  /\
             D,ro |- e  => v .
crl [IfR2] : D,ro |- If be Then e Else e' => v'
          if D,ro |- be => F  /\
             D,ro |- e' => v' .

crl [LocR] : D,ro |- let x = e in e' => v'
          if        D,ro |- e  => v /\
             D,ro[v / x] |- e' => v' .

*** call-by-value

crl [FunR] : D,ro |- F(el) => v
          if D,ro |- el => vl /\
             F(xl)<= e & D' := D /\
             D,ro[vl / xl] |- e => v .

crl [ExpLR] : D,ro |- e, el => v, vl
           if D,ro |- e => v /\
              D,ro |- el => vl .
```

```
  *** Evaluation semantics for boolean expressions

  rl [BCR1] : D,ro |- T => T .
  rl [BCR2] : D,ro |- F => F .

  rl [BVarR] : D,ro |- bx => ro(bx) .

  crl [BOpR] : D,ro |- be bop be' => Ap(bop,bv,bv')
            if D,ro |- be  => bv  /\
               D,ro |- be' => bv' .

  crl [NotR1] : D,ro |- Not be => F
             if D,ro |- be => T .
  crl [NotR2] : D,ro |- Not be => T
             if D,ro |- be => F .

  crl [EqR1] : D,ro |- Equal(e,e') => T
            if D,ro |- e  => v  /\
               D,ro |- e' => v .
  crl [EqR2] : D,ro |- Equal(e,e') => F
            if D,ro |- e  => v  /\
               D,ro |- e' => v' /\ v =/= v' .

endm
```

Obsérvese cómo con la condición `F(xl)<= e & D' := D` en la regla `FunR` se extrae del conjunto de declaraciones `D` la declaración correspondiente a la función `F`. La resolución de la condición mediante encaje de patrones, módulo asociatividad y conmutatividad, liga las variables `xl`, `e` y `D'`.

El módulo `EVALUATION` es un módulo admisible, directamente ejecutable en Maude 2.0. A continuación mostramos algunos ejemplos. En [Hen90] se ilustra esta semántica utilizando como ejemplo el programa $\langle Rem(\mathbf{3},\mathbf{5}), D\rangle$, donde D es la declaración de la función $Rem(x,y)$ que devuelve el resto de la división entera de y entre x. El conjunto de declaraciones `exDec1` incluye la declaración de esta función, tal y como se hace en [Hen90, página 63]. El siguiente comando evalúa el programa anterior:

```
Maude> rew exDec1, mt |- FV('Rem)(s(s(s(0))), s(s(s(s(s(0))))) .

rewrites: 240 in 0ms cpu (3ms real) (~ rewrites/second)
result Num: s(s(0))
```

Maude 2.0 tarda aproximadamente 3 milisegundos en reescribir el término en cuestión, ciertamente muy simple, y tarda unos 1,5 segundos en calcular el factorial de 9:

```
Maude> rew exDec1, mt |- FV('Fac)(s(s(s(s(s(s(s(s(s(0))))))))) .

rewrites: 409612 in 0ms cpu (1421ms real) (~ rewrites/second)
result Num: 362880
```

donde hemos adaptado la salida para mostrar el resultado en términos de enteros ordinarios. La mayor parte de este tiempo se consume debido a las ecuaciones de la operación `Ap`. Si modificamos la sintaxis para poder utilizar el tipo predefinido `Nat` como subtipo de `Num`, y utilizamos las operaciones predefinidas para definir la operación `Ap`, la ganancia es considerable, como muestran los siguientes ejemplos:

```
Maude> rew exDec1, mt |- FV('Fac)(9) .

rewrites: 418 in 0ms cpu (5ms real) (~ rewrites/second)
result NzNat: 362880

Maude> rew exDec1, mt |- FV('Fac)(42) .

rewrites: 1813 in 0ms cpu (27ms real) (~ rewrites/second)
result NzNat: 1405006117752879898543142606244511569936384000000000
```

También podemos pedir a Maude que trace el proceso de reescritura, mostrándonos en qué orden se van aplicando las reglas. Para poder mostrar el resultado, solo podemos trazar un ejemplo muy sencillo. La siguiente traza, modificada a mano para clarificar los pasos, muestra cómo se aplica la semántica de evaluación para calcular el factorial de 1. Los números utilizados para numerar las aplicaciones de las reglas corresponden a los usados para numerar los distintos juicios en el correspondiente árbol de derivación mostrado en la Figura 6.4. En este árbol hemos eliminado de cada juicio el conjunto de declaraciones de función, pues es siempre el mismo en toda la prueba.

```
Maude> rew exDec1, mt |- FV('Fac)(s(0)) .

*** rule CR                                                    (1)
exDec1,mt |- s(0)
--->
s(0)

*** rule VarR                                                  (2)
exDec1,V('x) = s(0) |- V('x)
--->
s(0)

*** rule CR                                                    (3)
exDec1,V('x) = s(0) |- 0
--->
0

*** rule EqR2                                                  (4)
exDec1,V('x) = s(0) |- Equal(V('x), 0)
--->
F
```

$$\dfrac{mt \vdash 1 \Longrightarrow_A 1 \ (1) \qquad \dfrac{\dfrac{x=1 \vdash x \Longrightarrow_A 1 \ (2) \qquad x=1 \vdash 0 \Longrightarrow_A 0 \ (3)}{x=1 \vdash \mathsf{Eq}(x,0) \Longrightarrow_B \mathsf{F} \ (4)} \qquad \dfrac{x=1 \vdash x \Longrightarrow_A 1 \ (5) \qquad \dfrac{\dfrac{x=1 \vdash x \Longrightarrow_A 1 \ (6) \qquad x=1 \vdash 1 \Longrightarrow_A 1 \ (7)}{x=1 \vdash x-1 \Longrightarrow_A 0 \ (8)} \qquad \dfrac{\dfrac{x=0 \vdash x \Longrightarrow_A 0 \ (9) \qquad x=0 \vdash 0 \Longrightarrow_A 0 \ (10)}{x=0 \vdash \mathsf{Eq}(x,0) \Longrightarrow_B \mathsf{T} \ (11)} \qquad x=0 \vdash 1 \Longrightarrow_A 1 \ (12)}{x=0 \vdash \mathsf{If...} \Longrightarrow_A 1 \ (13)}}{x=1 \vdash \mathtt{Fac}(x-1) \Longrightarrow_A 1 \ (14)}}{x=1 \vdash x * \mathtt{Fac}(x-1) \Longrightarrow_A 1 \ (15)}}{x=1 \vdash \mathsf{If}\ \mathsf{Eq}(x,0)\ \mathsf{Then}\ 1\ \mathsf{Else}\ x * \mathtt{Fac}(x-1) \Longrightarrow_A 1 \ (16)}}{mt \vdash \mathtt{Fac}(1) \Longrightarrow_A 1 \ (17)}$$

Figura 6.4: Árbol de derivación para $\mathtt{Fac}(1)$ (llamada por valor).

```
*** rule VarR                                                    (5)
exDec1,V('x) = s(0) |- V('x)
--->
s(0)

*** rule VarR                                                    (6)
exDec1,V('x) = s(0) |- V('x)
--->
s(0)

*** rule CR                                                      (7)
exDec1,V('x) = s(0) |- s(0)
--->
s(0)

*** rule OpR                                                     (8)
exDec1,V('x) = s(0) |- V('x) - s(0)
--->
Ap(-, s(0), s(0)) = 0

*** rule VarR                                                    (9)
exDec1,V('x) = 0 |- V('x)
--->
0

*** rule CR                                                      (10)
exDec1,V('x) = 0 |- 0
--->
0

*** rule EqR1                                                    (11)
exDec1,V('x) = 0 |- Equal(V('x), 0)
--->
T

*** rule CR                                                      (12)
exDec1,V('x) = 0 |- s(0)
--->
s(0)

*** rule IfR1                                                    (13)
exDec1,V('x) = 0 |- If Equal(V('x), 0) Then s(0)
                    Else V('x) * FV('Fac)(V('x) - s(0))
--->
s(0)

*** rule FunR                                                    (14)
exDec1,V('x) = s(0) |- FV('Fac)(V('x) - s(0))
--->
s(0)
```

```
*** rule OpR                                                          (15)
exDec1,V('x) = s(0) |- V('x) * FV('Fac)(V('x) - s(0))
--->
Ap(*, s(0), s(0)) = s(0)

*** rule IfR2                                                         (16)
exDec1,V('x) = s(0) |- If Equal(V('x), 0) Then s(0)
                       Else V('x) * FV('Fac)(V('x) - s(0))
--->
s(0)

*** rule FunR                                                         (17)
exDec1,mt |- FV('Fac)(s(0))
--->
s(0)

rewrites: 60 in 0ms cpu (199ms real) (~ rewrites/second)
result Num: s(0)
```

Antes dijimos que la regla FunR corresponde a una *llamada por valor*. La alternativa *llamada por nombre* deja los parámetros sin evaluar y simplemente los sustituye directamente en el cuerpo de la definición. La regla que describe este comportamiento es la siguiente:

$$\text{FunR}' \quad \frac{D, \rho \vdash e[e_1/x_1, \ldots, e_k/x_k] \Longrightarrow_A v}{D, \rho \vdash F(e_1, \ldots, e_k) \Longrightarrow_A v} \; F(x_1, \ldots, x_k) \Leftarrow e \text{ está en } D$$

donde se utiliza un operador de sustitución simultánea para sustituir, en la expresión e, las variables $x_1, \ldots, x_k$ por las expresiones en los parámetros actuales, $e_1, \ldots, e_k$.

La definición de la operación de sustitución $e[e'/v]$ tiene que tener en cuenta las peculiaridades de las variables libres y ligadas, para evitar la captura de variables y de modo que solo se sustituyan variables libres. Esta sustitución puede tener que introducir variables nuevas que no aparezcan ni en e ni en e'. El siguiente módulo funcional `SUBSTITUTION` define esta operación. En el caso de la sustitución simultánea $e[e_1/x_1, \ldots, e_k/x_k]$, suponemos que las variables que se sustituyen solo aparecen en e, por lo que se reduce a una serie de sustituciones simples.

La operación `new`, dado un conjunto (finito) de variables VS, devuelve una variable que no está en VS. Para obtener su valor se va probando con las variables $z1$, $z2$, etc. hasta que se encuentra una variable que no pertenezca al conjunto. Una nueva variable se necesita en el caso de una sustitución sobre un let que declare una variable que aparezca en la expresión que se sustituye.

```
fmod SUBSTITUTION is
  protecting FPL-SYNTAX .
  protecting STRING .
  protecting NUMBER-CONVERSION .

  sort VarSet .
```

```
op mt : -> VarSet .
subsort Var < VarSet .
op _U_ : VarSet VarSet -> VarSet [assoc comm id: mt] .
eq x U x = x . *** idempotency

*** FVar returns the set of free variables in an expression.
op FVar : Exp -> VarSet .
op FVar : BExp -> VarSet .
op FVar : ExpList -> VarSet .

op _in_ : Var VarSet -> Bool .
op _not-in_ : Var VarSet -> Bool .
op _\_ : VarSet VarSet -> VarSet .
op new : VarSet -> Var .
op new : VarSet Nat -> Var .
op newvar : Nat -> Var .

var n : Num .
vars x y x' : Var .
vars e e' e1 e2 : Exp .
var op : Op .
var F : FunVar .
var bx : BVar .
vars be be1 be2 : BExp .
var bop : BOp .
var el : ExpList .
vars VS VS' : VarSet .
var N : Nat .
var xl : VarList .

eq FVar(n) = mt .
eq FVar(x) = x .
eq FVar(e1 op e2) = FVar(e1) U FVar(e2) .
eq FVar(If be Then e1 Else e2) = FVar(be) U FVar(e1) U FVar(e2) .
eq FVar(let x = e in e') = (FVar(e') \ x) U FVar(e) .
eq FVar(F(el)) = FVar(el) .
eq FVar(e,el) = FVar(e) U FVar(el) .
eq FVar(T) = mt .
eq FVar(F) = mt .
eq FVar(Not be) = FVar(be) .
eq FVar(be1 bop be2) = FVar(be1) U FVar(be2) .
eq FVar(bx) = mt .
eq FVar(Equal(e1,e2)) = FVar(e1) U FVar(e2) .

eq x in mt = false .
eq x in (y U VS) = (x == y) or (x in VS) .

eq x not-in VS = not (x in VS) .
```

```
eq (mt \ VS') = mt .
eq (y U VS) \ VS' = if (y in VS') then VS \ VS'
                    else y U (VS \ VS') fi .

eq newvar(N) = V(qid("z" + string(N,10))) .

eq new(VS) = new(VS, 1) .
eq new(VS, N) = if newvar(N) not-in VS then newvar(N)
                else new(VS, N + 1) fi .

*** substitution of an expression for a variable

op _[_/_] : Exp Exp Var -> Exp .
op _[_/_] : BExp Exp Var -> BExp .
op _[_/_] : ExpList Exp Var -> ExpList .

eq y [e' / x] = if x == y then e' else y fi .

eq n [e' / x] = n .

eq (e1 op e2) [e' / x] = (e1 [e' / x]) op (e2 [e' / x]) .

eq (If be Then e1 Else e2) [e' / x] =
    If (be[e' / x]) Then (e1[e' / x]) Else (e2[e' / x]) .

eq (let x = e1 in e2) [e' / x] = let x = (e1 [e' / x]) in e2 .

ceq (let y = e1 in e2) [e' / x] =
     let y = (e1 [e' / x]) in (e2 [e' / x])
     if x =/= y /\ y not-in FVar(e') .

ceq (let y = e1 in e2) [e' / x] =
     let x' = (e1 [e' / x]) in ((e2[x' / y]) [e' / x])
     if x =/= y /\ y in FVar(e') /\
        x' := new(FVar(e') U FVar(e2)) .

eq F(el) [e' / x] = F(el [e' / x]) .

eq (e, el) [e' / x] =  (e[e' / x]), (el[e' / x]) .

eq T [e' / x] = T .
eq F [e' / x] = F .

eq bx [e' / x] = bx .

eq (be1 bop be2) [e' / x] = (be1 [e' / x]) bop (be2 [e' / x]) .

eq (Not be) [e' / x] = Not (be[e' / x]) .

eq Equal(e1,e2) [e' / x] = Equal(e1[e' / x],e2[e' / x]) .
```

```
  *** multiple simultaneous substitution

  op _[_/_] : Exp ExpList VarList -> Exp .

  eq e [e', el / x, xl] = (e [e' / x])[el / xl] .

endfm
```

Una vez definida la sustitución, podemos escribir la regla de reescritura que implementa la llamada por nombre:

```
  *** call-by-name
  crl [FunR'] : D,ro |- F(el) => v
             if F(xl)<= e & D' := D /\
                D,ro |- (e[el / xl]) => v .
```

Podemos probar esta semántica con el programa $\langle Rem(\mathbf{3},\mathbf{5}), D\rangle$, evaluado por el siguiente comando:

```
Maude> rew exDec1, mt |- FV('Rem)(s(s(s(0))), s(s(s(s(s(0)))))) .

rewrites: 234 in 0ms cpu (1ms real) (~ rewrites/second)
result Num: s(s(0))
```

También podemos trazar el mismo ejemplo que trazamos en la página 126, para comprobar cómo afecta el cambio realizado al cómputo de una expresión. El árbol de derivación correspondiente se muestra en la Figura 6.5.

```
Maude> rew exDec1, mt |- FV('Fac)(s(0)) .

*** rule CR                                                    (1)
exDec1,mt |- s(0)
--->
s(0)

*** rule CR                                                    (2)
exDec1,mt |- 0
--->
0

*** rule EqR2                                                  (3)
exDec1,mt |- Equal(s(0), 0)
--->
F
```

$$\dfrac{\dfrac{mt \vdash 1 \Longrightarrow_A 1\ (1) \quad mt \vdash 0 \Longrightarrow_A 0\ (2)}{mt \vdash \mathsf{Eq}(1,0) \Longrightarrow_B \mathsf{F}\ (3)} \quad mt \vdash 1 \Longrightarrow_A 1\ (4) \quad \dfrac{\dfrac{\dfrac{mt \vdash 1 \Longrightarrow_A 1\ (5) \quad mt \vdash 1 \Longrightarrow_A 1\ (6)}{mt \vdash 1-1 \Longrightarrow_A 0\ (7)} \quad mt \vdash 0 \Longrightarrow_A 0\ (8)}{mt \vdash \mathsf{Eq}(1-1,0) \Longrightarrow_B \mathsf{T}\ (9)} \quad mt \vdash 1 \Longrightarrow_A 1\ (10)}{mt \vdash \mathsf{If}\ \mathsf{Eq}(1-1,0)\ \mathsf{Then}\ 1\ \mathsf{Else}\ (1-1) * \mathtt{Fac}((1-1)-1) \Longrightarrow_A 1\ (11)}}{mt \vdash \mathtt{Fac}(1-1) \Longrightarrow_A 1\ (12)}$$

$$\dfrac{mt \vdash 1 * \mathtt{Fac}(1-1) \Longrightarrow_A 1\ (13)}{mt \vdash \mathsf{If}\ \mathsf{Eq}(1,0)\ \mathsf{Then}\ 1\ \mathsf{Else}\ 1 * \mathtt{Fac}(1-1) \Longrightarrow_A 1\ (14)}$$

$$mt \vdash \mathtt{Fac}(1) \Longrightarrow_A 1\ (15)$$

Figura 6.5: Árbol de derivación para $\mathtt{Fac}(1)$ (llamada por nombre).

```
*** rule CR                                                    (4)
exDec1,mt |- s(0)
--->
s(0)

*** rule CR                                                    (5)
exDec1,mt |- s(0)
--->
s(0)

*** rule CR                                                    (6)
exDec1,mt |- s(0)
--->
s(0)

*** rule OpR                                                   (7)
exDec1,mt |- s(0) - s(0)
--->
Ap(-, s(0), s(0)) = 0

*** rule CR                                                    (8)
exDec1,mt |- 0
--->
0

*** rule EqR1                                                  (9)
exDec1,mt |- Equal(s(0) - s(0), 0)
--->
T

*** rule CR                                                   (10)
exDec1,mt |- s(0)
--->
s(0)

*** rule IfR1                                                 (11)
exDec1,mt |- If Equal(s(0) - s(0), 0) Then s(0) Else
             (s(0) - s(0)) * FV('Fac)((s(0) - s(0)) - s(0))
--->
s(0)

*** rule FunR'                                                (12)
exDec1,mt |- FV('Fac)(s(0) - s(0))
--->
s(0)

*** rule OpR                                                  (13)
exDec1,mt |- s(0) * FV('Fac)(s(0) - s(0))
--->
Ap(*, s(0), s(0)) = s(0)
```

```
*** rule IfR2                                                    (14)
exDec1,mt |- If Equal(s(0), 0) Then s(0)
              Else s(0) * FV('Fac)(s(0) - s(0))
--->
s(0)

*** rule FunR'                                                   (15)
exDec1,mt |- FV('Fac)(s(0))
--->
s(0)

rewrites: 65 in 0ms cpu (158ms real) (~ rewrites/second)
result Num: s(0)
```

6.1.2. Semántica de computación

En esta sección veremos cómo implementar en Maude una semántica de computación (o paso corto) para *Fpl* que describe la secuencia de operaciones primitivas a la cual da lugar la evaluación de una expresión. Al igual que ocurre con la semántica de evaluación, serán necesarios los entornos de variables y las declaraciones. Los juicios de esta semántica para evaluar expresiones numéricas y booleanas son

$$D, \rho \vdash e \longrightarrow_A v \quad \text{y} \quad D, \rho \vdash be \longrightarrow_B bv.$$

Las reglas semánticas que definen estos dos juicios se presentan en las Figuras 6.6 y 6.7, respectivamente.

Para la implementación vamos a utilizar el módulo `FPL-SYNTAX` presentado al comienzo de la Sección 6.1 aunque con algunas modificaciones. Para poder expresar con facilidad y claridad la regla FunRc de la Figura 6.6 vamos a necesitar que el operador de construcción de listas de expresiones tenga como elemento identidad la lista vacía, como veremos a continuación:

```
  subsort Exp < ExpList .
  op nil : -> ExpList .
  op _,_ : ExpList ExpList -> ExpList [assoc id: nil prec 30] .
```

Los módulos `AP`, `ENV` y `SUBSTITUTION` se utilizan sin modificación. El siguiente módulo `COMPUTATION` contiene la implementación de las reglas semánticas. Obsérvese como la primera regla `FunRc` expresa la elección no determinista de uno de los argumentos para ser reescrito, mediante un encaje de patrones de la lista de argumentos con el patrón `el,e,el'` modulo asociatividad, conmutatividad y elemento neutro. Este patrón incluye también los casos de uno o dos argumentos, que se obtienen haciendo vacía alguna de las listas en juego.

```
mod COMPUTATION is
  protecting AP .
  protecting ENV .
```

VarRc $$\frac{}{D, \rho \vdash x \longrightarrow_A \rho(x)}$$

OpRc $$\frac{}{D, \rho \vdash v\ op\ v' \longrightarrow_A Ap(op, v, v')}$$

$$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash e\ op\ e' \longrightarrow_A e''\ op\ e'} \qquad \frac{D, \rho \vdash e' \longrightarrow_A e''}{D, \rho \vdash e\ op\ e' \longrightarrow_A e\ op\ e''}$$

IfRc $$\frac{D, \rho \vdash be \longrightarrow_B be'}{D, \rho \vdash \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e' \longrightarrow_A \mathsf{If}\ be'\ \mathsf{Then}\ e\ \mathsf{Else}\ e'}$$

$$\frac{}{D, \rho \vdash \mathsf{If\ T\ Then}\ e\ \mathsf{Else}\ e' \longrightarrow_A e} \qquad \frac{}{D, \rho \vdash \mathsf{If\ F\ Then}\ e\ \mathsf{Else}\ e' \longrightarrow_A e'}$$

LocRc $$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \longrightarrow_A \mathsf{let}\ x = e''\ \mathsf{in}\ e'}$$

$$\frac{}{D, \rho \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e' \longrightarrow_A e'[v/x]}$$

FunRc $$\frac{D, \rho \vdash e_i \longrightarrow_A e_i'}{D, \rho \vdash F(e_1, \ldots, e_i, \ldots, e_k) \longrightarrow_A F(e_1, \ldots, e_i', \ldots, e_k)}$$

$$\frac{}{D, \rho \vdash F(v_1, \ldots, v_k) \longrightarrow_A e[v_1/x_1, \ldots, v_k/x_k]}\ F(x_1, \ldots, x_k) \Leftarrow e \text{ está en } D$$

Figura 6.6: Semántica de computación para *Fpl*, $\longrightarrow_A$.

BVarRc $$\frac{}{D, \rho \vdash bx \longrightarrow_B \rho(bx)}$$

BOpRc $$\frac{}{D, \rho \vdash bv\ bop\ bv' \longrightarrow_B Ap(bop, bv, bv')}$$

$$\frac{D, \rho \vdash be \longrightarrow_B be''}{D, \rho \vdash be\ bop\ be' \longrightarrow_B be''\ bop\ be'} \qquad \frac{D, \rho \vdash be' \longrightarrow_B be''}{D, \rho \vdash be\ bop\ be' \longrightarrow_B be\ bop\ be''}$$

NotRc $$\frac{D, \rho \vdash be \longrightarrow_B be'}{D, \rho \vdash \mathsf{Not}\ be \longrightarrow_B \mathsf{Not}\ be'}$$

$$\frac{}{D, \rho \vdash \mathsf{Not\ T} \longrightarrow_B \mathsf{F}} \qquad \frac{}{D, \rho \vdash \mathsf{Not\ F} \longrightarrow_B \mathsf{T}}$$

EqRc $$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash \mathsf{Equal}(e, e') \longrightarrow_B \mathsf{Equal}(e'', e')}$$

$$\frac{D, \rho \vdash e' \longrightarrow_A e''}{D, \rho \vdash \mathsf{Equal}(e, e') \longrightarrow_B \mathsf{Equal}(e', e'')}$$

$$\frac{v = v'}{D, \rho \vdash \mathsf{Equal}(v, v') \longrightarrow_B \mathsf{T}} \qquad \frac{v \neq v'}{D, \rho \vdash \mathsf{Equal}(v, v') \longrightarrow_B \mathsf{F}}$$

Figura 6.7: Semántica de computación para expresiones booleanas, $\longrightarrow_B$.

```
protecting SUBSTITUTION .

sort Statement .
subsorts Num Boolean < Statement .
op _,_|-_ : Dec ENV Exp -> Statement [prec 40] .
op _,_|-_ : Dec ENV BExp -> Statement [prec 40] .

vars D D' : Dec .
var ro : ENV .
var x : Var .
var bx : BVar .
vars v v' : Num .
vars bv bv' : Boolean .
var op : Op .
vars e e' e'' : Exp .
vars el el' : ExpList .
vars be be' be'' : BExp .
var xl : VarList .
var vl : NumList .
var bop : BOp .
var F : FunVar .

*** computation semantics for Fpl

rl [VarRc] : D,ro |- x => ro(x) .

rl  [OpRc1] : D,ro |- v op v' => Ap(op,v,v') .
crl [OpRc2] : D,ro |- e op e' => e'' op e'
           if D,ro |- e  => e'' .
crl [OpRc3] : D,ro |- e op e' => e op e''
           if D,ro |- e'  => e'' .

crl [IfRc1] : D,ro |- If be Then e Else e' => If be' Then e Else e'
           if D,ro |- be => be' .
rl  [IfRc2] : D,ro |- If T Then e Else e' => e .
rl  [IfRc3] : D,ro |- If F Then e Else e' => e' .

crl [LocRc1] : D,ro |- let x = e in e' => let x = e'' in e'
            if D,ro |- e  => e'' .
rl  [LocRc2] : D,ro |- let x = v in e' => e'[v / x] .

crl [FunRc1] : D,ro |- F(el,e,el') => F(el,e',el')
            if D,ro |- e => e' .
crl [FunRc2] : D,ro |- F(vl) => e[vl / xl]
            if F(xl)<= e & D' := D .

*** computation semantics for boolean expressions

rl [BVarRc] : D,ro |- bx => ro(bx) .
```

```
  rl  [BOpRc1] : D,ro |- bv bop bv' => Ap(bop,bv,bv') .
  crl [BOpRc2] : D,ro |- be bop be' => be'' bop be'
              if D,ro |- be  => be'' .
  crl [BOpRc3] : D,ro |- be bop be' => be bop be''
              if D,ro |- be'  => be'' .

  crl [NotRc1] : D,ro |- Not be => Not be'
              if D,ro |- be => be' .
  rl  [NotRc2] : D,ro |- Not T => F .
  rl  [NotRc3] : D,ro |- Not F => T .

  crl [EqRc1] : D,ro |- Equal(e,e') => Equal(e'',e')
             if D,ro |- e => e'' .
  crl [EqRc2] : D,ro |- Equal(e,e') => Equal(e,e'')
             if D,ro |- e' => e'' .
  crl [EqRc3] : D,ro |- Equal(v,v') => T
             if v == v' .
  crl [EqRc4] : D,ro |- Equal(v,v') => F
             if v =/= v' .


  *** reflexive, transitive closure

  op _,_|=_ : Dec ENV Exp -> Statement [prec 40] .
  op _,_|=_ : Dec ENV BExp -> Statement [prec 40] .

  rl  [zero] : D,ro |= v => v .  *** no step
  crl [more] : D,ro |= e => v
            if D,ro |- e => e'   *** one step
            /\ D,ro |= e' => v . *** all the rest

  rl  [zero] : D,ro |= bv => bv .  *** no step
  crl [more] : D,ro |= be => bv
            if D,ro |- be => be'   *** one step
            /\ D,ro |= be' => bv . *** all the rest

endm
```

Hemos definido también el cierre transitivo y reflexivo de las transiciones $\longrightarrow_A$ y $\longrightarrow_B$. Obsérvese que si hubiéramos utilizado los mismos constructores `_,_|-_` para los términos que aparecen a la izquierda de las reglas `zero` y `more`, estas reglas podrían utilizarse para intentar resolver la primera condición de las reglas `more`, generando bucles infinitos. Hemos evitado el problema utilizando los nuevos operadores `_,_|=_`. Esta es una técnica similar a la que hemos utilizado en la Sección 5.2 (página 111) al introducir operadores ficticios para definir el cierre transitivo y reflexivo de las transiciones CCS.

Podemos utilizar esta implementación de la semántica de computación para evaluar la expresión $Rem(\mathbf{3},\mathbf{5})$, considerada en la sección anterior. Obsérvese que se utiliza el cierre transitivo y reflexivo `_,_|=_`. Si utilizaramos la relación `_,_|-_` solo conseguiríamos dar un paso, ya que al aplicar una regla se obtiene una expresión, que no encaja con el lado

izquierdo de ninguna regla. Este hecho también se muestra a continuación:

```
Maude> rew exDec1, mt |= FV('Rem)(s(s(s(0))), s(s(s(s(s(0))))))  .

rewrites: 181 in 0ms cpu (1ms real) (~ rewrites/second)
result Num: s(s(0))

Maude> rew exDec1, mt |- FV('Fac)(s(s(s(s(0))))) .

rewrites: 19 in 0ms cpu (0ms real) (~ rewrites/second)
result Exp: If Equal(s(s(s(s(0)))), 0) Then s(0)
            Else s(s(s(s(0)))) * FV('Fac)(s(s(s(s(0)))) - s(0))
```

Podemos también observar de nuevo la traza producida al evaluar el factorial de 1. Para simplificar solo mostraremos las transiciones de las relaciones $\longrightarrow_A$ y $\longrightarrow_B$, y hemos eliminado las aplicaciones de la regla `zero` (1 vez) y de la regla `more` (8 veces), al final de la traza.

```
Maude> rew exDec1, mt |= FV('Fac)(s(0)) .

*** rule FunRc2
exDec1,mt |- FV('Fac)(s(0))
--->
If Equal(s(0), 0) Then s(0) Else s(0) * FV('Fac)(s(0)) - s(0))

*** rule EqRc4
exDec1,mt |- Equal(s(0), 0)
--->
F

*** rule IfRc1
exDec1,mt |- If Equal(s(0), 0) Then s(0) Else s(0) * FV('Fac)(s(0) - s(0))
--->
If F Then s(0) Else s(0) * FV('Fac)(s(0) - s(0))

*** rule IfRc3
exDec1,mt |- If F Then s(0) Else s(0) * FV('Fac)(s(0) - s(0))
--->
s(0) * FV('Fac)(s(0) - s(0))

*** rule OpRc1
exDec1,mt |- s(0) - s(0)
--->
Ap(-, s(0), s(0)) = 0

*** rule FunRc1
exDec1,mt |- FV('Fac)(s(0) - s(0))
--->
FV('Fac)(nil,0,nil)
```

```
*** rule OpRc3
exDec1,mt |- s(0) * FV('Fac)(s(0) - s(0))
--->
s(0) * FV('Fac)(0)

*** rule FunRc2
exDec1,mt |- FV('Fac)(0)
--->
If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0))

*** rule OpRc3
exDec1,mt |- s(0) * FV('Fac)(0)
--->
s(0) * (If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0)))

*** rule EqRc3
exDec1,mt |- Equal(0, 0)
--->
T

*** rule IfRc1
exDec1,mt |- If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0))
--->
If T Then s(0) Else (0).Num * FV('Fac)(0 - s(0))

*** rule OpRc3
exDec1,mt |- s(0) * (If Equal(0, 0) Then s(0) Else 0 * FV('Fac)(0 - s(0)))
--->
s(0) * (If T Then s(0) Else 0 * FV('Fac)(0 - s(0)))

*** rule IfRc2
exDec1,mt |- If T Then s(0) Else 0 * FV('Fac)(0 - s(0))
--->
s(0)

*** rule OpRc3
exDec1,mt |- s(0) * (If T Then s(0) Else 0 * FV('Fac)(0 - s(0)))
--->
s(0) * s(0)

*** rule OpRc1
exDec1,mt |- s(0) * s(0)
--->
Ap(*, s(0), s(0)) = s(0)

rewrites: 68 in 0ms cpu (255ms real) (~ rewrites/second)
result Num: s(0)
```

Las reglas semánticas de la Figura 6.6 utilizan la sustitución sintáctica de variables por valores en una expresión. Pero los entornos fueron introducidos con el propósito de

LocRc
$$\frac{D, \rho \vdash e \longrightarrow_A e''}{D, \rho \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' \longrightarrow_A \mathsf{let}\ x = e''\ \mathsf{in}\ e'}$$
$$\frac{D, \rho[v/x] \vdash e \longrightarrow_A e'}{D, \rho \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow_A \mathsf{let}\ x = v\ \mathsf{in}\ e'}$$
$$\frac{}{D, \rho \vdash \mathsf{let}\ x = v\ \mathsf{in}\ v' \longrightarrow_A v'}$$

FunRc
$$\frac{D, \rho \vdash e_i \longrightarrow_A e_i'}{D, \rho \vdash F(e_1, \ldots, e_i, \ldots, e_k) \longrightarrow_A F(e_1, \ldots, e_i', \ldots, e_k)}$$
$$\frac{}{D, \rho \vdash F(v_1, \ldots, v_k) \longrightarrow_A \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ \ldots\ \mathsf{let}\ x_k = v_k\ \mathsf{in}\ e}\ F(x_1, \ldots, x_k) \Leftarrow e \text{ está en } D$$

Figura 6.8: Modificación de las reglas para no utilizar sustitución.

almacenar las asociaciones entre variables y valores, por lo que podría ser preferible no ocultar parte de esta misión con el uso de las sustituciones. Una forma de suprimir las sustituciones es definir las reglas semánticas LocRc y FunRc como se hace en la Figura 6.8.

Las siguientes reglas de reescritura implementan estas nuevas reglas semánticas. Hemos utilizado la operación auxiliar `buildLet` para construir la expresión resultado en la segunda regla FunRc.

```
op buildLet : VarList NumList Exp -> Exp .

eq buildLet(nil, nil, e) = e .
eq buildLet(x, v, e) = let x = v in e .
eq buildLet((x, xl), (v, vl), e) = let x = v in buildLet(xl, vl, e) .

crl [LocRc1] : D,ro |- let x = e in e' => let x = e'' in e'
            if D,ro |- e  => e'' .
crl [LocRc2] : D,ro |- let x = v in e => let x = v in e'
            if D,ro[v / x] |-  e => e' .
rl  [LocRc3] : D,ro |- let x = v in v' => v' .

crl [FunRc1] : D,ro |- F(el,e,el') => F(el,e',el')
            if D,ro |- e => e' .
crl [FunRc2] : D,ro |- F(vl) => buildLet(xl, vl, e)
            if F(xl)<= e & D' := D .
```

6.1.3. Máquina abstracta para *Fpl*

En esta sección presentaremos una semántica operacional concreta para *Fpl* basada en una máquina abstracta. Obtenemos así un intérprete formal del lenguaje que se ejecuta en una máquina virtual. Los estados de esta máquina serán tuplas $\langle S, env, C\rangle$, donde S es una pila de valores, env es un entorno (una lista finita de asociaciones entre variables

1. Categorías sintácticas

$$\begin{array}{rcl@{\qquad}rcl} st & \in & States & C & \in & Control \\ S & \in & Stack & env & \in & Env \\ a & \in & Assoc & cons & \in & Constants \\ F & \in & FunVar & v & \in & Num \\ bv & \in & Boolean & & & \end{array}$$

2. Definiciones

$$\begin{array}{rcl} st & ::= & \langle S, env, C\rangle \\ S & ::= & \varepsilon \mid v.S \mid bv.S \\ C & ::= & \varepsilon \mid e.C \mid cons.C \\ cons & ::= & op \mid bop \mid \langle x, e\rangle \mid \mathsf{if}(e, e') \mid \mathsf{not} \mid \mathsf{equal} \mid F \mid \mathsf{pop} \\ env & ::= & \varepsilon \mid a.env \\ a & ::= & (x, v) \mid (bx, bv) \mid (F, (x, e)) \\ v & ::= & \mathbf{n} \\ bv & ::= & \mathsf{T} \mid \mathsf{F} \end{array}$$

Figura 6.9: Estados de la máquina abstracta para *Fpl*.

y valores, en este caso) y C es una secuencia de control. La sintaxis abstracta de estos estados se muestra en la Figura 6.9.

Para describir el funcionamiento de la máquina es suficiente definir una relación $\longrightarrow$ entre estados,

$$\langle S, env, C\rangle \longrightarrow \langle S', env', C'\rangle.$$

Las reglas semánticas que definen esta relación se muestran en las Figuras 6.10 y 6.11.

Las reglas Varm y Funm2 utilizan en sus premisas predicados como $env, x \vdash v$ para indicar que v es el valor al que la variable x está ligada en el entorno env. No mostramos la definición obvia de estos predicados, que serán implementados por medio de las funciones `lookup` en el módulo `ABS-MACHINE-SEMANTICS`.

El siguiente módulo funcional `ABS-MACHINE-SYNTAX` define la sintaxis de los estados de la máquina abstracta.

```
fmod ABS-MACHINE-SYNTAX is
  protecting AP .

  *** states of the abstract machine for Fpl

  sorts Constants Assoc Env Stack Control States .

  op <_,_,_> : Stack Env Control -> States [prec 60] .

  op mtS : -> Stack .
  op _._ : Num Stack -> Stack .
  op _._ : Boolean Stack -> Stack .

  op mtC : -> Control .
  op _._ : Exp Control -> Control [prec 50] .
```

$$\text{Opm1} \quad \langle S, env, e\ op\ e'.C\rangle \longrightarrow \langle S, env, e.e'.op.C\rangle$$

$$\langle S, env, be\ bop\ be'.C\rangle \longrightarrow \langle S, env, be.be'.bop.C\rangle$$

$$\text{Ifm1} \quad \langle S, env, \mathsf{If}\ be\ \mathsf{Then}\ e\ \mathsf{Else}\ e'.C\rangle \longrightarrow \langle S, env, be.\mathsf{if}(e, e').\mathsf{equal}.C\rangle$$

$$\text{Locm1} \quad \langle S, env, \mathsf{let}\ x = e\ \mathsf{in}\ e'.C\rangle \longrightarrow \langle S, env, e.\langle x, e'\rangle.C\rangle$$

$$\text{Funm1} \quad \langle S, env, F(e).C\rangle \longrightarrow \langle S, env, e.F.C\rangle$$

$$\text{Notm1} \quad \langle S, env, \mathsf{Not}\ be.C\rangle \longrightarrow \langle S, env, be.\mathsf{not}.C\rangle$$

$$\text{Eqm1} \quad \langle S, env, \mathsf{Equal}(e, e').C\rangle \longrightarrow \langle S, env, e.e'.\mathsf{equal}.C\rangle$$

Figura 6.10: Reglas de análisis para la máquina abstracta.

$$\text{Opm2} \quad \langle v'.v.S, env, op.C\rangle \longrightarrow \langle Ap(op, v, v').S, env, C\rangle$$

$$\langle bv'.bv.S, env, bop.C\rangle \longrightarrow \langle Ap(bop, bv, bv').S, env, C\rangle$$

$$\text{Varm} \quad \frac{env, x \vdash v}{\langle S, env, x.C\rangle \longrightarrow \langle v.S, env, C\rangle} \qquad \frac{env, bx \vdash bv}{\langle S, env, bx.C\rangle \longrightarrow \langle bv.S, env, C\rangle}$$

$$\text{Valm} \quad \langle S, env, v.C\rangle \longrightarrow \langle v.S, env, C\rangle \qquad \langle S, env, bv.C\rangle \longrightarrow \langle bv.S, env, C\rangle$$

$$\text{Notm2} \quad \langle \mathsf{T}.S, env, \mathsf{not}.C\rangle \longrightarrow \langle \mathsf{F}.S, env, C\rangle \qquad \langle \mathsf{F}.S, env, \mathsf{not}.C\rangle \longrightarrow \langle \mathsf{T}.S, env, C\rangle$$

$$\text{Eqm2} \quad \frac{v = v'}{\langle v.v'.S, env, \mathsf{equal}.C\rangle \longrightarrow \langle \mathsf{T}.S, env, C\rangle}$$

$$\frac{v \neq v'}{\langle v.v'.S, env, \mathsf{equal}.C\rangle \longrightarrow \langle \mathsf{F}.S, env, C\rangle}$$

$$\text{Ifm2} \quad \langle \mathsf{T}.S, env, \mathsf{if}(e, e').C\rangle \longrightarrow \langle S, env, e.C\rangle$$

$$\langle \mathsf{F}.S, env, \mathsf{if}(e, e').C\rangle \longrightarrow \langle S, env, e'.C\rangle$$

$$\text{Funm2} \quad \frac{env, F \vdash (x, e)}{\langle v.S, env, F.C\rangle \longrightarrow \langle S, (x, v).env, e.\mathsf{pop}.C\rangle}$$

$$\text{Locm2} \quad \langle v.S, env, \langle x, e\rangle.C\rangle \longrightarrow \langle S, (x, v).env, e.\mathsf{pop}.C\rangle$$

$$\text{Pop} \quad \langle S, (x, v).env, \mathsf{pop}.C\rangle \longrightarrow \langle S, env, C\rangle$$

Figura 6.11: Reglas de aplicación para la máquina abstracta.

```
  op _._ : BExp Control -> Control [prec 50] .
  op _._ : Constants Control -> Control [prec 50] .

  subsort Op BOp < Constants .

  op <_,_> : Var Exp -> Constants .
  op if : Exp Exp -> Constants .
  op not : -> Constants .
  op equal : -> Constants .
  op pop : -> Constants .
  subsort FunVar < Constants .

  op mtE : -> Env .
  subsort Assoc < Env .
  op _._ : Env Env -> Env [assoc id: mtE] .

  op `(_,_`) : Var Num -> Assoc .
  op `(_,_`) : BVar Boolean -> Assoc .
  op `(_,_,_`) : FunVar Var Exp -> Assoc .
endfm
```

El módulo `ABS-MACHINE-SEMANTICS` implementa las reglas de análisis y de aplicación de la máquina abstracta. Obsérvese que las reglas en este módulo *no* necesitan reescrituras en las condiciones, ya que las reglas semánticas de las Figuras 6.10 y 6.11 no tienen premisas con transiciones.

```
mod ABS-MACHINE-SEMANTICS is
  protecting ABS-MACHINE-SYNTAX .

  op lookup : Env Var -> Num .
  op lookup : Env BVar -> Boolean .
  op lookup : Env FunVar -> Constants .

  vars x x' : Var .
  vars v v' : Num .
  vars bx bx' : BVar .
  vars bv bv' : Boolean .
  vars FV FV' : FunVar .
  vars e e' : Exp .
  var op : Op .
  var bop : BOp .
  var be be' : BExp .
  var env : Env .
  var S : Stack .
  var C : Control .

  eq lookup((x',v) . env, x) = if x == x' then v else lookup(env,x) fi .
  eq lookup((bx, bv) . env, x) = lookup(env, x) .
  eq lookup((FV,x',e) . env, x) = lookup(env, x) .
```

```
eq lookup((x,v) . env, bx) = lookup(env,bx) .
eq lookup((bx',bv) . env, bx) = if bx == bx' then bv else lookup(env,bx) fi .
eq lookup((FV,x',e) . env, bx) = lookup(env,bx) .

eq lookup((x,v) . env, FV) = lookup(env,FV) .
eq lookup((bx',bv) . env, FV) = lookup(env,FV) .
eq lookup((FV',x',e) . env, FV) =
   if FV == FV' then < x', e > else lookup(env,FV) fi .


*** Analysis rules for the abstract machine

rl [Opm1]  : < S, env, e op e' . C > => < S, env, (e . e' . op . C) > .
rl [Opm1'] : < S, env, be bop be' . C > => < S, env, be . be' . bop . C > .

rl [Notm1] : < S, env, Not be . C > => < S, env, be . not . C > .

rl [Eqm1] : < S, env, Equal(e,e') . C > => < S, env, e . e' . equal . C > .

rl [Ifm1] : < S, env, If be Then e Else e' . C >
         => < S, env, be . if(e,e') . C > .

rl [Funm1] : < S, env, FV(e) . C > => < S, env, e . FV . C > .

rl [Locm1] : < S, env, let x = e in e' . C > => < S, env, e . < x, e' > . C > .


*** Application rules for the abstract machine

rl [Opm2]  : < v' . v . S, env, op . C > => < Ap(op, v, v') . S, env, C > .
rl [Opm2'] : < bv' . bv . S, env, bop . C >
          => < Ap(bop, bv, bv') . S, env, C > .

crl [Varm]  : < S, env, x . C > => < v . S, env, C >
              if v := lookup(env, x) .
crl [Varm'] : < S, env, bx . C > => < bv . S, env, C >
              if bv := lookup(env, bx) .

rl [Valm]  : < S, env, v . C > => < v . S, env, C > .
rl [Valm'] : < S, env, bv . C > => < bv . S, env, C > .

rl [Notm2]  : < T . S, env, not . C > => < F . S, env, C > .
rl [Notm2'] : < F . S, env, not . C > => < T . S, env, C > .

crl [Eqm2]  : < v . v' . S, env, equal . C > => < T . S, env, C > if v == v' .
crl [Eqm2'] : < v . v' . S, env, equal . C > => < F . S, env, C > if v =/= v' .

rl [Ifm2 ] : < T . S, env, if(e,e') . C > => < S, env, e . C > .
rl [Ifm2'] : < F . S, env, if(e,e') . C > => < S, env, e' . C > .
```

```
  crl [Funm2] : < v . S, env, FV . C > => < S, (x, v) . env, e . pop . C >
   if < x, e > := lookup(env,FV) .

  rl [Locm2] : < v . S, env, < x, e > . C > => < S, (x,v) . env, e . pop . C > .

  rl [Pop] : < S, (x,v) . env, pop . C > => < S, env, C > .

endm
```

Calculando el factorial de 1 podemos ver cómo funciona la máquina abstracta. Las reescrituras en esta semántica son completamente deterministas, ya que no hay reescrituras en las condiciones, no hay dos reglas con la misma parte izquierda, y todas las reescrituras se hacen al nivel más alto. Por eso hemos modificado la traza producida por Maude, para mostrar esta secuencialidad. Además, hemos escrito `dec1` en vez de la declaración completa de la función `Fac`, que sí mostramos en el comando `rew` introducido en Maude.

```
Maude> rew < mtS, (FV('Fac), V('x), If Equal(V('x),0) Then s(0)
                                    Else V('x) * FV('Fac)(V('x) - s(0))),
             FV('Fac)(s(0)) . mtC > .

< mtS, dec1, FV('Fac)(s(0)) . mtC >
--->  *** rule Funm1
< mtS, dec1, s(0) . FV('Fac) . mtC >
--->  *** rule Valm
< s(0) . mtS, dec1, FV('Fac) . mtC >
--->  *** rule Funm2
< mtS, (V('x),s(0)) . dec1, If Equal(V('x), 0) Then s(0)
                            Else V('x) * FV('Fac)(V('x) - s(0)) . pop . mtC >
--->  *** rule Ifm1
< mtS, (V('x),s(0)) . dec1, Equal(V('x), 0) . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Eqm1
< mtS, (V('x),s(0)) . dec1, V('x) . 0 . equal . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Varm
< s(0) . mtS, (V('x),s(0)) . dec1, 0 . equal . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Valm
< 0 . s(0) . mtS, (V('x),s(0)) . dec1, equal . if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . mtC >
--->  *** rule Eqm2'
< F . mtS, (V('x),s(0)) . dec1, if(s(0), V('x) * FV('Fac)(V('x) - s(0))) .
    pop . mtC >
--->  *** rule Ifm2'
< mtS, (V('x),s(0)) . dec1, V('x) * FV('Fac)(V('x) - s(0)) . pop . mtC >
--->  *** rule Opm1
< mtS, (V('x),s(0)) . dec1, V('x) . FV('Fac)(V('x) - s(0)) . * . pop . mtC >
--->  *** rule Varm
< s(0) . mtS, (V('x),s(0)) . dec1, FV('Fac)(V('x) - s(0)) . * . pop . mtC >
```

```
--->  *** rule Funm1
< s(0) . mtS, (V('x),s(0)) . dec1, V('x) - s(0) . FV('Fac) . * . pop . mtC >
--->  *** rule Opm1
< s(0) . mtS,(V('x),s(0)) . dec1, V('x) . s(0) . - . FV('Fac) . * . pop . mtC >
--->  *** rule Varm
< s(0) . s(0) . mtS, (V('x),s(0)) . dec1, s(0) . - . FV('Fac) . * . pop . mtC >
--->  *** rule Valm
< s(0) . s(0) . s(0) . mtS, (V('x),s(0)) . dec1, - . FV('Fac) . * . pop . mtC >
--->  *** rule Opm2
< 0 . s(0) . mtS, (V('x),s(0)) . dec1, FV('Fac) . * . pop . mtC >
--->  *** rule Funm2
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, If Equal(V('x), 0) Then
    s(0) Else V('x) * FV('Fac)(V('x) - s(0)) . pop . * . pop . mtC >
--->  *** rule Ifm1
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, Equal(V('x), 0) .
    if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Eqm1
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, V('x) . 0 . equal .
    if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Varm
< 0 . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, 0 .
    equal . if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Valm
< 0 . 0 . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, equal .
    if(s(0), V('x) * FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Eqm2
< T . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, if(s(0), V('x) *
    FV('Fac)(V('x) - s(0))) . pop . * . pop . mtC >
--->  *** rule Ifm2
< s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, s(0) . pop . * . pop . mtC >
--->  *** rule Valm
< s(0) . s(0) . mtS, (V('x),0) . (V('x),s(0)) . dec1, pop . * . pop . mtC >
--->  *** rule Pop
< s(0) . s(0) . mtS, (V('x),s(0)) . dec1, * . pop . mtC >
--->  *** rule Opm2
< s(0) . mtS, (V('x),s(0)) . dec1, pop . mtC >
--->  *** rule Pop
< s(0) . mtS, dec1, mtC >

rewrites: 55 in 0ms cpu (152ms real) (~ rewrites/second)
result States: < s(0) . mtS, dec1, mtC >
```

6.2. El lenguaje *WhileL*

En esta sección presentamos dos semánticas operacionales (de evaluación y de computación) para un lenguaje de programación imperativo simple denominado *WhileL* en [Hen90]. Un programa es una secuencia de comandos, cada uno de los cuales puede modificar la memoria, vista como una colección de direcciones donde se almacenan valores.

1. Categorías sintácticas

$$\begin{array}{rcl@{\qquad\qquad}rcl} p & \in & \mathit{Prog} & C & \in & \mathit{Com} \\ op & \in & \mathit{Op} & bop & \in & \mathit{BOp} \\ e & \in & \mathit{Exp} & x & \in & \mathit{Var} \\ be & \in & \mathit{BExp} & bx & \in & \mathit{BVar} \\ n & \in & \mathit{Num} & & & \end{array}$$

2. Definiciones

$$\begin{array}{rcl} p & ::= & C \\ C & ::= & \mathsf{skip} \mid x := e \mid C'; C'' \mid \mathsf{If}\ be\ \mathsf{Then}\ C'\ \mathsf{Else}\ C''\mid \mathsf{While}\ be\ \mathsf{Do}\ C' \\ op & ::= & + \mid - \mid * \\ e & ::= & n \mid x \mid e'\ op\ e'' \\ bop & ::= & \mathit{And} \mid \mathit{Or} \\ be & ::= & bx \mid \mathsf{T} \mid \mathsf{F} \mid be'\ bop\ be'' \mid \mathsf{Not}\ be' \mid \mathsf{Equal}(e, e') \end{array}$$

Figura 6.12: Sintaxis abstracta de *WhileL*.

La sintaxis abstracta del lenguaje *WhileL* se muestra en la Figura 6.12, y se implementa en el módulo `WHILE-SYNTAX`. Obsérvese que la estructura de la signatura corresponde a la estructura de la gramática que define la sintaxis abstracta del lenguaje.

```
fmod WHILE-SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp Com Prog .

  op V : Qid -> Var .
  subsort Var < Exp .
  subsort Num < Exp .

  op 0 : -> Num .
  op s : Num -> Num .

  ops + - * : -> Op .

  op ___ : Exp Op Exp -> Exp [prec 20] .

  op BV : Qid -> BVar .
  subsort BVar < BExp .
  subsort Boolean < BExp .

  ops T F : -> Boolean .
  ops And Or : -> BOp .
  op ___ : BExp BOp BExp -> BExp [prec 20] .
  op Not_ : BExp -> BExp [prec 15] .
  op Equal : Exp Exp -> BExp .

  op skip : -> Com .
  op _:=_ : Var Exp -> Com [prec 30] .
```

```
  op _;_ : Com Com -> Com [assoc prec 40] .
  op If_Then_Else_ : BExp Com Com -> Com [prec 50] .
  op While_Do_ : BExp Com -> Com [prec 60] .

  subsort Com < Prog .

endfm
```

6.2.1. Semántica de evaluación

La semántica de evaluación de *WhileL* viene dada por tres relaciones: $\Longrightarrow_A$, $\Longrightarrow_B$ y $\Longrightarrow_C$, correspondientes a cada una de las categorías sintácticas *Exp*, *BExp* y *Com*. También se utilizan entornos para guardar el valor de las variables. Sin embargo, aquí las variables juegan un papel muy diferente al que jugaban en el lenguaje *Fpl*; ahora representan direcciones de memoria y, como dijimos antes, el cómputo procede modificando el contenido de esta memoria.

La relación de evaluación $\Longrightarrow_A$ para expresiones numéricas toma un par formado por una expresión y una memoria y devuelve un valor, resultado de evaluar la expresión sobre la memoria. Lo mismo ocurre con la relación $\Longrightarrow_B$ para expresiones booleanas. Su definición se muestra en la Figura 6.13.

El módulo `EVALUATION-EXP` implementa estas dos relaciones.

```
mod EVALUATION-EXP is
  protecting ENV .
  protecting AP .

  sort Statement .
  subsorts Num Boolean < Statement .

  op <_,_> : Exp ENV -> Statement .
  op <_,_> : BExp ENV -> Statement .

  var n : Num .
  var x : Var .
  var st : ENV .
  vars e e' : Exp .
  var op : Op .
  vars v v' : Num .
  var bx : BVar .
  vars bv bv' : Boolean .
  var bop : BOp .
  vars be be' : BExp .

  *** Evaluation semantics for expressions

  rl [CR] : < n, st > => n .

  rl [VarR] : < x, st > =>  st(x) .
```

$$\text{CR}\quad \frac{}{(\mathbf{n}, s) \Longrightarrow_A \mathbf{n}}$$

$$\text{VarR}\quad \frac{}{(x, s) \Longrightarrow_A s(x)}$$

$$\text{OpR}\quad \frac{(e, s) \Longrightarrow_A v \qquad (e', s) \Longrightarrow_A v'}{(e\ op\ e', s) \Longrightarrow_A Ap(op, v, v')}$$

$$\text{BCR}\quad \frac{}{(\mathsf{T}, s) \Longrightarrow_B \mathsf{T}} \qquad \frac{}{(\mathsf{F}, s) \Longrightarrow_B \mathsf{F}}$$

$$\text{BVarR}\quad \frac{}{(bx, s) \Longrightarrow_B s(bx)}$$

$$\text{BOpR}\quad \frac{(be, s) \Longrightarrow_B bv \quad (be', s) \Longrightarrow_B bv'}{(be\ bop\ be', s) \Longrightarrow_B Ap(bop, bv, bv')}$$

$$\text{EqR}\quad \frac{(e, s) \Longrightarrow_A v \quad (e', s) \Longrightarrow_A v}{(\mathsf{Equal}(e, e'), s) \Longrightarrow_B \mathsf{T}} \qquad \frac{(e, s) \Longrightarrow_A v \quad (e', s) \Longrightarrow_A v'}{(\mathsf{Equal}(e, e'), s) \Longrightarrow_B \mathsf{F}} \quad v \neq v'$$

$$\text{NotR}\quad \frac{(be, s) \Longrightarrow_B \mathsf{T}}{(\mathsf{Not}\ be, s) \Longrightarrow_B \mathsf{F}} \qquad \frac{(be, s) \Longrightarrow_B \mathsf{F}}{(\mathsf{Not}\ be, s) \Longrightarrow_B \mathsf{T}}$$

Figura 6.13: Semántica de evaluación para *WhileL*, $\Longrightarrow_A$ y $\Longrightarrow_B$.

```
  crl [OpR] : < e op e', st > => Ap(op,v,v')
           if < e, st > => v /\
              < e', st > => v' .

  rl [BCR1] : < T, st > => T .
  rl [BCR2] : < F, st > => F .

  rl [BVarR] : < bx, st > =>  st(bx) .

  crl [OpR] : < be bop be', st > => Ap(bop,bv,bv')
           if < be, st > => bv /\
              < be', st > => bv' .

  crl [EqR1] : < Equal(e,e'), st > => T
            if < e, st > => v /\
               < e', st > => v .
  crl [EqR2] : < Equal(e,e'), st > => F
            if < e, st > => v /\
               < e', st > => v' /\ v =/= v' .

  crl [Not1] : < Not be, st > => F
            if < be, st > => T .
  crl [Not2] : < Not be, st > => T
            if < be, st > => F .
endm
```

La relación de evaluación para instrucciones $\Longrightarrow_C$ toma un par formado por una instrucción y una memoria y devuelve una nueva memoria. Intuitivamente, la memoria devuelta es el resultado de modificar la memoria inicial por medio de la instrucción ejecutada. Así, un juicio

$$(C, s) \Longrightarrow_C s'$$

significa que cuando se ejecuta la instrucción C sobre la memoria s, la ejecución termina y el estado final de la memoria es s'. La definición de esta relación se muestra en la Figura 6.14, mientras que el módulo `EVALUATION-WHILE` implementa la relación $\Longrightarrow_C$.

```
mod EVALUATION-WHILE is
  protecting EVALUATION-EXP .

  subsort ENV < Statement .

  op <_,_> : Com ENV -> Statement .

  var x : Var .
  vars st st' st'' : ENV .
  var e : Exp .
  var v : Num .
  var be : BExp .
  vars C C' : Com .
```

$$\text{SkipR}\quad \frac{}{(\mathsf{skip}, s) \Longrightarrow_C s}$$

$$\text{AsR}\quad \frac{(e, s) \Longrightarrow_A v}{(x := e, s) \Longrightarrow_C s[v/x]}$$

$$\text{ComR}\quad \frac{(C, s) \Longrightarrow_C s' \quad (C', s') \Longrightarrow_C s''}{C; C' \Longrightarrow_C s''}$$

$$\text{IfR}\quad \frac{(be, s) \Longrightarrow_B \mathsf{T} \quad (C, s) \Longrightarrow_C s'}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s) \Longrightarrow_C s'} \qquad \frac{(be, s) \Longrightarrow_B \mathsf{F} \quad (C', s) \Longrightarrow_C s'}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s) \Longrightarrow_C s'}$$

$$\text{WhileR}\quad \frac{(be, s) \Longrightarrow_B \mathsf{F}}{(\mathsf{While}\ be\ \mathsf{Do}\ C, s) \Longrightarrow_C s} \qquad \frac{(be, s) \Longrightarrow_B \mathsf{T} \quad (C'; \mathsf{While}\ be\ \mathsf{Do}\ C, s) \Longrightarrow_C s'}{(\mathsf{While}\ be\ \mathsf{Do}\ C, s) \Longrightarrow_C s'}$$

Figura 6.14: Semántica de evaluación para *WhileL*, $\Longrightarrow_C$.

```
  *** Evaluation semantics for WhileL

  crl [AsR] : < x := e, st > => st[v / x]
           if < e, st > => v .

  rl [SkipR] : < skip, st > => st .

  crl [IfR1] : < If be Then C Else C', st > => st'
            if < be, st > => T /\
               < C, st > => st' .
  crl [IfR2] : < If be Then C Else C', st > => st'
            if < be, st > => F /\
               < C', st > => st' .

  crl [ComR] : < C ; C', st > => st''
            if < C, st > => st' /\
               < C', st' > => st'' .

  crl [WhileR1] : < While be Do C, st > => st
               if < be, st > => F .
  crl [WhileR2] : < While be Do C, st > => st'
               if < be, st > => T /\
                  < C ; (While be Do C), st > => st' .
endm
```

Como ejemplo de aplicación de estas reglas consideraremos el siguiente programa

$z := 0$;
While Not (Equal$(x, 0)$) Do
    $z := z + y$;
    $x := x - 1$

que calcula el producto $x * y$ y guarda el resultado en z. Lo ejecutamos comenzando con una memoria s en la que $s(x) = 2$, $s(y) = 3$ y $s(z) = 1$.

```
Maude> rew
     < V('z) := 0 ;
      (While Not Equal(V('x), 0) Do
          V('z) := V('z) + V('y) ;
          V('x) := V('x) - s(0)),
      V('x) = s(s(0)) V('y) = s(s(s(0))) V('z) = s(0) > .

rewrites: 267 in 0ms cpu (630ms real) (~ rewrites/second)
result ENV: V('x) = 0 V('y) = s(s(s(0))) V('z) = s(s(s(s(s(0)))))
```

6.2.2. Semántica de computación

Las instrucciones básicas de *WhileL* son las asignaciones, que modifican una memoria modificando el valor asociado a una variable. Una semántica de computación para *WhileL* debe describir las operaciones básicas que cada instrucción puede realizar e indicar un orden entre ellas. El juicio

$$(C, s) \longrightarrow_C (C', s')$$

significa que la instrucción C puede ejecutar una operación básica, que cambia la memoria de s a s', siendo C' el resto que todavía queda por ejecutarse de la instrucción C.

En esta sección asumimos que no estamos interesados en indicar cómo se computan las expresiones numéricas y booleanas, por lo que no definiremos relaciones $\longrightarrow_A$ y $\longrightarrow_B$, utilizando en su lugar las relaciones de evaluación $\Longrightarrow_A$ y $\Longrightarrow_B$ de la sección anterior. Las reglas que definen la relación $\longrightarrow_C$ para instrucciones se muestran en la Figura 6.15, donde $(C, s)\surd$ indica que la ejecución de la instrucción C ha terminado. La definición del predicado de terminación $\surd$ se muestra en la Figura 6.16.

La implementación de estas reglas semánticas se muestra a continuación en el módulo `COMPUTATION-WHILE`. El predicado de terminación se ha implementado mediante reglas que reescriben un par formado por una instrucción y una memoria a la constante `Tick`. El uso de reglas de reescritura, y no de ecuaciones, es necesario ya que la definición del predicado hace uso de transiciones en las premisas de las reglas IfRt de la Figura 6.16.

```
mod COMPUTATION-WHILE is
  protecting EVALUATION-EXP .

  op <_,_> : Com ENV -> Statement .
```

$$\text{AsRc} \quad \frac{(e,s) \Longrightarrow_A v}{(x := e, s) \longrightarrow_C (\mathsf{skip}, s[v/x])}$$

$$\text{ComRc} \quad \frac{(C,s) \longrightarrow_C (C'', s')}{(C; C', s) \longrightarrow_C (C''; C', s')} \qquad \frac{(C,s)\surd \qquad (C', s) \longrightarrow_C (C'', s')}{(C; C', s) \longrightarrow_C (C'', s')}$$

$$\text{IfRc} \quad \frac{(be,s) \Longrightarrow_B \mathsf{T} \qquad (C,s) \longrightarrow_C (C'', s')}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s) \longrightarrow_C (C'', s')}$$

$$\frac{(be,s) \Longrightarrow_B \mathsf{F} \qquad (C',s) \longrightarrow_C (C'', s')}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s) \longrightarrow_C (C'', s')}$$

$$\text{WhileRc} \quad \frac{(be,s) \Longrightarrow_B \mathsf{F}}{(\mathsf{While}\ be\ \mathsf{Do}\ C, s) \longrightarrow_C (\mathsf{skip}, s)}$$

$$\frac{(be,s) \Longrightarrow_B \mathsf{T}}{(\mathsf{While}\ be\ \mathsf{Do}\ C, s) \longrightarrow_C (C; \mathsf{While}\ be\ \mathsf{Do}\ C, s)}$$

Figura 6.15: Semántica de computación para *WhileL*, $\longrightarrow_C$.

$$\text{Skipt} \quad \frac{}{(\mathsf{skip}, s)\surd}$$

$$\text{ComRt} \quad \frac{(C,s)\surd \qquad (C',s)\surd}{(C; C', s)\surd}$$

$$\text{IfRt} \quad \frac{(be,s) \Longrightarrow_B \mathsf{T} \qquad (C,s)\surd}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s)\surd}$$

$$\frac{(be,s) \Longrightarrow_B \mathsf{F} \qquad (C',s)\surd}{(\mathsf{If}\ be\ \mathsf{Then}\ C\ \mathsf{Else}\ C', s)\surd}$$

Figura 6.16: Predicado de terminación para *WhileL*.

```
  sort Statement2 .
  op `(_,_`) : Com ENV -> Statement2 .
  op Tick : -> Statement2 .

  var x : Var .
  vars st st' : ENV .
  var e : Exp .
  var v : Num .
  var be : BExp .
  vars C C' C'' : Com .

  *** Computation semantics for WhileL

  crl [AsRc] : < x := e, st > => < skip, st[v / x] >
           if < e, st > => v .

  crl [IfRc1] : < If be Then C Else C', st > => < C'', st' >
            if < be, st > => T /\
               < C, st > => < C'', st' > /\ C =/= C'' .
  crl [IfRc2] : < If be Then C Else C', st > => < C'', st' >
            if < be, st > => F /\
               < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [ComRc1] : < C ; C', st > => < C'' ; C', st' >
             if < C, st > => < C'', st' > /\ C =/= C'' .
  crl [ComRc2] : < C ; C', st > => < C'', st' >
             if ( C, st ) => Tick /\
                < C', st > => < C'', st' > /\ C' =/= C'' .

  crl [WhileRc1] : < While be Do C, st > => < skip, st >
               if < be, st > => F .
  crl [WhileRc2] : < While be Do C, st > => < C ; (While be Do C), st >
               if < be, st > => T .


  *** Termination predicate for WhileL

  rl [Skipt] : ( skip, st ) => Tick .

  crl [IfRt1] : ( If be Then C Else C', st ) => Tick
            if < be, st > => T /\
               ( C, st ) => Tick .
  crl [IfRt2] : ( If be Then C Else C', st ) => Tick
            if < be, st > => F /\
               ( C', st ) => Tick .

  crl [ComRt] : ( C ; C', st ) => Tick
            if ( C, st ) => Tick /\
               ( C', st ) => Tick .
endm
```

1. Categorías sintácticas

$$\begin{array}{rcl} p & \in & Prog \\ C & \in & Com \\ GC & \in & GuardCom \\ e & \in & Exp \\ be & \in & BExp \end{array}$$

2. Definiciones

$$\begin{array}{rcl} p & ::= & C \\ C & ::= & \mathsf{skip} \mid x := e \mid C'; C'' \mid \mathsf{if}\ GC\ \mathsf{fi} \mid \mathsf{do}\ GC\ \mathsf{od} \\ GC & ::= & be \rightarrow C \mid GC' \ \square\ GC'' \end{array}$$

Figura 6.17: Sintaxis abstracta de *GuardL*.

Podemos ejecutar el programa ejemplo de la sección anterior.

```
Maude> rew
     < V('z) := 0 ;
      (While Not Equal(V('x),0) Do
        V('z) := V('z) + V('y) ;
        V('x) := V('x) - s(0)),
      V('x) = s(s(0)) V('y) = s(s(s(0))) V('z) = s(0) > .

rewrites: 299 in 0ms cpu (826ms real) (~ rewrites/second)
result Statement:
  < skip, V('x) = 0 V('y) = s(s(s(0))) V('z) = s(s(s(s(s(0))))) >
```

6.2.3. El lenguaje *GuardL*

En [Hen90] se presenta también una generalización del lenguaje *WhileL* que permite no determinismo, denominada *GuardL*. Este lenguaje fue definido originalmente por Dijkstra en [Dij76], donde se presentaba como un lenguaje conveniente para el desarrollo de programas y su verificación. El no determinismo se consideraba muy útil, ya que permitía al diseñador del programa delegar ciertas decisiones al implementador o compilador.

La sintaxis abstracta del lenguaje *GuardL* se muestra en la Figura 6.17. Aparece una nueva categoría sintáctica de *instrucciones guardadas*, cuyos elementos tienen la forma general

$$be_1 \rightarrow C_1 \ \square\ \ldots\ \square\ be_k \rightarrow C_k.$$

El siguiente módulo implementa la sintaxis de *GuardL* teniendo la misma estructura que la correspondiente gramática.

```
fmod GUARDL-SYNTAX is
  protecting QID .

  sorts Var Num Op Exp BVar Boolean BOp BExp Com GuardCom Prog .
```

```
  op V : Qid -> Var .
  subsort Var < Exp .
  subsort Num < Exp .

  op z : -> Num .
  op s : Num -> Num .

  ops + - * : -> Op .

  op ___ : Exp Op Exp -> Exp [prec 20] .

  op BV : Qid -> BVar .
  subsort BVar < BExp .
  subsort Boolean < BExp .

  ops T F : -> Boolean .
  ops And Or : -> BOp .
  op ___ : BExp BOp BExp -> BExp [prec 20] .
  op Not_ : BExp -> BExp [prec 15] .
  op Equal : Exp Exp -> BExp .

  op skip : -> Com .
  op _:=_ : Var Exp -> Com [prec 30] .
  op _;_ : Com Com -> Com [assoc prec 40] .
  op if_fi : GuardCom -> Com [prec 50] .
  op do_od : GuardCom -> Com [prec 60] .

  op _->_ : BExp Com -> GuardCom [prec 42] .
  op _[]_ : GuardCom GuardCom -> GuardCom [assoc prec 45] .

  subsort Com < Prog .

endfm
```

Se dice que la expresión booleana be_i *guarda* a la instrucción C_i correspondiente, que solo se ejecutará si el control "pasa a través de la guarda be_i", es decir, esta se evalúa a cierto. En la instrucción if GC fi, solo se ejecutará una de las instrucciones asociada a una guarda cierta. Si ninguna de las guardas es cierta, se considera que se ha producido un error de ejecución. De la misma forma, la instrucción do GC od es una generalización de la instrucción While. La instrucción guardada GC se ejecuta de forma repetida, mientras al menos una de las guardas sea cierta. La terminación tiene lugar cuando todas las guardas son falsas. Estas ideas intuitivas se formalizan en la semántica de computación de la Figura 6.18, que define dos relaciones de transición, $\longrightarrow_C$ y $\longrightarrow_{GC}$, y utiliza la semántica de evaluación de la Figura 6.13 para las expresiones aritméticas y booleanas.

Para formalizar el hecho de que todas las guardas booleanas de una instrucción guardada son falsas, se necesita un predicado de fallo. Este se define de forma inductiva en la Figura 6.19. También se utiliza un predicado de terminación, como en el caso de *WhileL*,

$$\text{AsRc} \quad \frac{(e,s) \Longrightarrow_A v}{(x := e, s) \longrightarrow_C (\mathsf{skip}, s[v/x])}$$

$$\text{ComRc} \quad \frac{(C,s) \longrightarrow_C (C'',s')}{(C;C',s) \longrightarrow_C (C'';C',s')} \qquad \frac{(C,s)\surd \qquad (C',s) \longrightarrow_C (C'',s')}{(C;C',s) \longrightarrow_C (C'',s')}$$

$$\text{IfRc} \quad \frac{(GC,s) \longrightarrow_{GC} (C,s)}{(\mathsf{if}\ GC\ \mathsf{fi}, s) \longrightarrow_C (C,s)}$$

$$\text{DoRc} \quad \frac{(GC,s) \longrightarrow_{GC} (C,s)}{(\mathsf{do}\ GC\ \mathsf{od}, s) \longrightarrow_C (C; \mathsf{do}\ GC\ \mathsf{od}, s)} \qquad \frac{(GC,s)\ \mathit{fails}}{(\mathsf{do}\ GC\ \mathsf{od}, s) \longrightarrow_C (\mathsf{skip}, s)}$$

$$\text{GCRc1} \quad \frac{(be,s) \Longrightarrow_B \mathsf{T}}{(be \rightarrow C, s) \longrightarrow_{GC} (C,s)}$$

$$\text{GCRc2} \quad \frac{(GC_1,s) \longrightarrow_{GC} (C,s)}{(GC_1 \,\square\, GC_2, s) \longrightarrow_{GC} (C,s)} \qquad \frac{(GC_2,s) \longrightarrow_{GC} (C,s)}{(GC_1 \,\square\, GC_2, s) \longrightarrow_{GC} (C,s)}$$

Figura 6.18: Semántica de computación para *GuardL*, $\longrightarrow_C$ y $\longrightarrow_{GC}$.

$$\text{Skipt} \quad \frac{(be,s) \Longrightarrow_B \mathsf{F}}{(be \rightarrow C, s)\ \mathit{fails}}$$

$$\text{ComRt} \quad \frac{(GC,s)\ \mathit{fails} \qquad (GC',s)\ \mathit{fails}}{(GC \,\square\, GC', s)\ \mathit{fails}}$$

Figura 6.19: Predicado de fallo para *GuardL*.

aunque en este caso es aún más sencillo, y se define en la Figura 6.20.

El siguiente módulo `GUARDL-COMPUTATION` implementa la semántica de computación del lenguaje *GuardL*.

```
mod GUARDL-COMPUTATION is
  protecting ENV .
  protecting AP .
  protecting EVALUATION-EXP .

  op <_,_> : Com ENV -> Statement .
  op <_,_> : GuardCom ENV -> Statement .

  sort Statement2 .
  op `(_,_`) : Com ENV -> Statement2 .
  op Tick : -> Statement2 .
  op `(_,_`) : GuardCom ENV -> Statement2 .
  op fails : -> Statement2 .
```

$$\text{Skipt} \quad \frac{}{(\mathsf{skip}, s)\surd}$$

$$\text{ComRt} \quad \frac{(C, s)\surd \qquad (C', s)\surd}{(C; C', s)\surd}$$

Figura 6.20: Predicado de terminación para *GuardL*.

```
var x : Var .
vars st st' : ENV .
var e : Exp .
var op : Op .
vars v v' : Num .
var be : BExp .
vars C C' C'' : Com .
vars GC GC' : GuardCom .


*** Computation semantics for GuardL

crl [AsRc] : < x := e, st > => < skip, st[v / x] >
          if < e, st > => v .

crl [ifRc] : < if GC fi, st > => < C, st >
          if < GC, st > => < C, st > .

crl [ComRc1] : < C ; C', st > => < C'' ; C', st' >
            if < C, st > => < C'', st' > /\ C =/= C'' .
crl [ComRc2] : < C ; C', st > => < C'', st' >
            if ( C, st ) => Tick /\
               < C', st > => < C'', st' > /\ C' =/= C'' .

crl [doRc1] : < do GC od, st > => < C ; (do GC od), st >
           if < GC, st > => < C, st > .
crl [doRc2] : < do GC od, st > => < skip, st >
           if ( GC, st ) => fails .


crl [GCRc1] : < be -> C, st > => < C, st >
           if < be, st > => T .

crl [GCRc2] : < GC [] GC', st > => < C, st >
           if < GC, st > => < C, st > .
crl [GCRc2] : < GC [] GC', st > => < C, st >
           if < GC', st > => < C, st > .
```

```
  *** Failure predicate for GuardL

  crl [IfRf1] : ( be -> C, st ) => fails
              if < be, st > => F .

  crl [IfRf2] : ( GC [] GC', st ) => fails
              if ( GC, st )  => fails /\
                 ( GC', st ) => fails .


  *** Termination predicate for GuardL

  rl [Skipt] : ( skip, st ) => Tick .

  crl [ComRt] : ( C ; C', st ) => Tick
              if ( C, st ) => Tick /\
                 ( C', st ) => Tick .

endm
```

Los predicados de fallo y terminación también se han implementado mediante reglas, como en la sección anterior. Sin embargo, en este caso, no es necesario utilizar reglas de reescritura para el predicado de terminación, ya que las únicas reescrituras en las condiciones son las propias que definen el predicado. Así, podemos definir este predicado mediante una operación booleana, aunque especificando solo los casos verdaderos.

```
  op `(_,_`)Tick : Com ENV -> Bool .

  eq ( skip, st )Tick = true .
  ceq ( C ; C', st )Tick = true
   if ( C, st )Tick  /\  ( C', st )Tick .
```

Utilizando este predicado, la regla `ComRc2` tendría que modificarse de la siguiente manera:

```
  crl [ComRc2] : < C ; C', st > => < C'', st' >
               if ( C, st )Tick /\
                  < C', st > => < C'', st' > /\ C' =/= C'' .
```

Para ilustrar esta semántica podemos ejecutar la siguiente instrucción, utilizada en [Hen90, página 133]:

$$
\begin{array}{l}
\mathsf{do} \\
\quad x > 0 \to x := x - 1 \; ; \; y := y + 1 \\
\quad \square \\
\quad x > 2 \to x := x - 2 \; ; \; y := y + 1 \\
\mathsf{od}
\end{array}
$$

Si empezamos a ejecutar esta instrucción en una memoria s en la que $s(x) = 5$ y $s(y) = 0$, y suponiendo reglas apropiadas para la operación $>$, se pueden alcanzar tres estados distintos. Para pedir a Maude que nos muestre todos los estados finales alcanzables, utilizamos el comando `search` (con la versión `=>+` para que pueda dar varios pasos hasta encontrar el término buscado).

```
Maude> search
      < do
            V('x) > 0       -> V('x) := V('x) - s(0) ; V('y) := V('y) + s(0)
         []
            V('x) > s(s(0)) -> V('x) := V('x) - s(s(0)) ; V('y) := V('y) + s(0)
        od,
      V('x) = s(s(s(s(s(0))))) V('y) = 0 >
  =>+ < skip, st:ENV > .

Solution 1 (state 25)
st --> V('x) = 0 V('y) = s(s(s(s(s(0)))))

Solution 2 (state 29)
st --> V('x) = 0 V('y) = s(s(s(s(0))))

Solution 3 (state 38)
st --> V('x) = 0 V('y) = s(s(s(0)))

No more solutions.
```

6.3. El lenguaje Mini-ML

En esta sección implementamos la semántica de evaluación (o semántica natural) del lenguaje funcional Mini-ML utilizado por Kahn en [Kah87]. La sintaxis abstracta, tal y como se define en [Kah87], se presenta en la Figura 6.21. Obsérvese cómo esta presentación de la sintaxis está más cercana a la signatura de una especificación algebraica. La sintaxis define un λ-cálculo extendido con productos, if, let y letrec. En una expresión λP.E, P puede ser un patrón. En Mini-ML un patrón puede ser un identificador, como la variable x, o una pareja de patrones (P_1, P_2), como el patrón $(x, (y, z))$.

La sintaxis de Mini-ML se implementa por medio de los módulos siguientes:

```
fmod NATS is
  sort Nats .
  op 0 : -> Nats .
  op s : Nats -> Nats .
endfm

fmod TRUTH-VAL is
  sort TruthVal .
  ops true false : -> TruthVal .
endfm
```

sorts
EXP, IDENT, PAT, NULLPAT
subsorts
EXP ⊃ NULLPAT, IDENT PAT ⊃ NULLPAT, IDENT
constructors
Patterns

pairpat	:	PAT×PAT	→	PAT
nullpat	:		→	NULLPAT

Expressions

number	:		→	EXP
false	:		→	EXP
true	:		→	EXP
ident	:		→	IDENT
lambda	:	PAT×EXP	→	EXP
if	:	EXP×EXP×EXP	→	EXP
mlpair	:	EXP×EXP	→	EXP
apply	:	EXP×EXP	→	EXP
let	:	PAT×EXP×EXP	→	EXP
letrec	:	PAT×EXP×EXP	→	EXP

Figura 6.21: Sintaxis abstracta de Mini-ML.

```
fmod VAR is
  protecting QID .
  sort Var .
  op id : Qid -> Var .
endfm

fmod MINI-ML-SYNTAX is
  protecting NATS .
  protecting TRUTH-VAL .
  protecting VAR .

  sorts Exp Value Pat NullPat Lambda .

  subsorts NullPat Var < Pat .
  op `(`) : -> NullPat .
  op `(_,_`) : Pat Pat -> Pat .
  op `(_,_`) : Var Var -> Var .

  subsorts TruthVal Nats < Value .
  op `(_,_`) : Value Value -> Value .

  subsorts Value Var Lambda < Exp .
  op s : Exp -> Exp .
  op _+_ : Exp Exp -> Exp [prec 20] .
  op not : Exp -> Exp .
  op _and_ : Exp Exp -> Exp .
```

```
  op if_then_else_ : Exp Exp Exp -> Exp [prec 22] .
  op `(_,_`) : Exp Exp -> Exp .
  op __ : Exp Exp -> Exp [prec 20] .
  op \_._ : Pat Exp -> Lambda [prec 15] .
  op let_=_in_ : Pat Exp Exp -> Exp [prec 25] .
  op letrec_=_in_ : Pat Exp Exp -> Exp [prec 25] .
endfm
```

En [Kah87] se define la semántica de Mini-ML utilizando juicios de la forma

$$\rho \vdash \text{E} \Rightarrow \alpha$$

donde E es una expresión de Mini-ML, ρ es un entorno, y α es el resultado de la evaluación de E en ρ. Con las funciones se trabaja como con cualquier otro valor; por ejemplo, pueden ser pasadas como parámetros a otras funciones, o devueltas como valor de una expresión.

Los valores semánticos son

- valores enteros,
- valores booleanos *true* y *false*,
- clausuras de la forma $[\![\lambda \text{P.E}, \rho]\!]$, donde P es un patrón, E es una expresión y ρ un entorno.
- pares de valores semánticos de la forma (α, β) (los cuales, naturalmente, pueden ser a su vez pares, dando lugar a árboles de valores semánticos).

Las reglas semánticas de Mini-ML, tal y como se presentan en [Kah87], se muestran en la Figura 6.22.

La implementación de los entornos y de las reglas semánticas, haciendo excepción de la regla del operador letrec que presenta problemas y se verá más adelante, se muestra en los módulos siguientes:

```
fmod ENV is
  including MINI-ML-SYNTAX .
  sort Pair .
  op _->_ : Pat Value -> Pair [prec 10] .
  sort Env .
  subsort Pair < Env .
  op nil : -> Env .
  op _;_ : Env Env -> Env [assoc id: nil prec 20] .
  op Clos : Lambda Env -> Value .
endfm

mod MINI-ML-SEMANTICS is
  including ENV .

  sort Statement .
```

$$\rho \vdash \text{number N} \Rightarrow \text{N}$$

$$\rho \vdash \text{true} \Rightarrow \mathit{true}$$

$$\rho \vdash \text{false} \Rightarrow \mathit{false}$$

$$\rho \vdash \lambda\text{P.E} \Rightarrow [\![\lambda\text{P.E}, \rho]\!]$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident I} \mapsto \alpha}{\rho \vdash \text{ident I} \Rightarrow \alpha}$$

$$\frac{\rho \vdash \text{E}_1 \mapsto \mathit{true} \qquad \rho \vdash \text{E}_2 \Rightarrow \alpha}{\rho \vdash \text{if E}_1 \text{ then E}_2 \text{ else E}_3 \Rightarrow \alpha}$$

$$\frac{\rho \vdash \text{E}_1 \mapsto \mathit{false} \qquad \rho \vdash \text{E}_3 \Rightarrow \alpha}{\rho \vdash \text{if E}_1 \text{ then E}_2 \text{ else E}_3 \Rightarrow \alpha}$$

$$\frac{\rho \vdash \text{E}_1 \Rightarrow \alpha \qquad \rho \vdash \text{E}_2 \Rightarrow \beta}{\rho \vdash (\text{E}_1, \text{E}_2) \Rightarrow (\alpha, \beta)}$$

$$\frac{\rho \vdash \text{E}_1 \Rightarrow [\![\lambda\text{P.E}, \rho_1]\!] \qquad \rho \vdash \text{E}_2 \Rightarrow \alpha \qquad \rho \cdot \text{P} \mapsto \alpha \vdash \text{E} \Rightarrow \beta}{\rho \vdash \text{E}_1\ \text{E}_2 \Rightarrow \beta}$$

$$\frac{\rho \vdash \text{E}_2 \Rightarrow \alpha \qquad \rho \cdot \text{P} \mapsto \alpha \vdash \text{E}_1 \Rightarrow \beta}{\rho \vdash \text{let P} = \text{E}_2 \text{ in E}_1 \Rightarrow \beta}$$

$$\frac{\rho \cdot \text{P} \mapsto \alpha \vdash \text{E}_2 \Rightarrow \alpha \qquad \rho \cdot \text{P} \mapsto \alpha \vdash \text{E}_1 \Rightarrow \beta}{\rho \vdash \text{letrec P} = \text{E}_2 \text{ in E}_1 \Rightarrow \beta}$$

Figura 6.22: Semántica de evaluación para Mini-ML.

```
op _|-_ : Env Exp -> Statement [prec 40] .
op _|-val-of_ : Env Var -> Statement [prec 40] .

subsort Value < Statement .

vars R0 R01 : Env .
vars N M : Nats .
vars P P1 P2 : Pat .
vars E E1 E2 E3 : Exp .
vars I X : Qid .
vars A B C : Value .

rl [number] : R0 |- N => N .

crl [add] : R0 |- E1 + E2 => C  if
            R0 |- E1 => A  /\  R0 |- E2 => B  /\  C := sum(A,B) .

op sum : Nats Nats -> Nats .
eq sum(0,N) = N .
eq sum(s(N),M) = s(sum(N,M)) .

rl [true] : R0 |- true => true .
rl [false] : R0 |- false => false .


rl [lambda] : R0 |- \ P . E => Clos( \ P . E, R0) .

crl [id] : R0 |- id(I) => A
        if R0 |-val-of id(I) => A .

crl [if] : R0 |- if E1 then E2 else E3 => A if
           R0 |- E1 => true /\ R0 |- E2 => A .
crl [if] : R0 |- if E1 then E2 else E3 => A if
           R0 |- E1 => false /\ R0 |- E3 => A .

crl [pair] : R0 |- ( E1, E2 ) => ( A, B ) if
             R0 |- E1 => A  /\  R0 |- E2 => B .

crl [app] : R0 |- E1 E2 => B if
            R0 |- E1 => Clos( \ P . E, R01)  /\
            R0 |- E2 => A  /\
           (R01 · P -> A ) |- E => B .

crl [let] : R0 |- let P = E2 in E1 => B if
            R0 |- E2 => A  /\  (R0 · P -> A) |- E1 => B .

*** set VAL_OF

rl [val_of] : R0 · id(I) -> A |-val-of id(I) => A .
```

```
  crl [val_of] : RO · id(X) -> B |-val-of id(I) => A
              if X =/= I /\ RO |- id(I) => A .

  crl [val_of] : RO · (P1, P2) -> (A, B) |-val-of id(I) => C
              if RO · P1 -> A · P2 -> B  |-val-of id(I) => C .
endm
```

Evaluaremos ahora algunas de las expresiones utilizadas como ejemplos en [Kah87]. Por ejemplo, podemos ver la estructura de bloques del lenguaje y el uso de patrones con la siguiente expresión que se evalúa a 3.

$$\begin{array}{l}\mathsf{let}\ (x, y) = (2, 3)\\ \mathsf{in\ let}\ (x, y) = (y, x)\ \mathsf{in}\ x\end{array}$$

```
Maude> rew nil |- let ( id('x), id('y) ) = ( s(s(0)), s(s(s(0))) )
                  in let ( id('x), id('y) ) = ( id('y), id('x) ) in id('x) .

result Nats: s(s(s(0)))
```

También podemos utilizar funciones de orden superior, como en

$$\begin{array}{l}\mathsf{let}\ succ = \lambda x.x + 1\\ \mathsf{in\ let}\ twice = \lambda f.\lambda x.(f(f\ x))\\ \quad \mathsf{in}\ ((twice\ succ)\ 0)\end{array}$$

```
Maude> rew
 nil |- let id('succ) = (\ id('x) . (id('x) + s(0)))
        in let id('twice) = (\ id('f) . (\ id('x) . (id('f) (id('f) id('x)))))
           in (( id('twice) id('succ)) 0) .

result Nats: s(s(0))
```

La regla semántica del operador letrec no se puede implementar de una forma directa, ya que la regla

```
  crl [letrec] : RO |- letrec P = E2 in E1 => B if
                 (RO · P -> A) |- E2 => A  /\  (RO · P -> A) |- E1 => B .
```

no resulta admisible debido a que en el lado izquierdo de la primera condición de reescritura la variable `A` es una variable *nueva*.

Esto hace que debamos replantearnos la presentación de la regla semántica que define este operador. Ahora bien, trabajando con llamada por valor, como hacemos aquí, solo tiene sentido permitir definiciones recursivas de λ-abstracciones, por lo que asumiremos que este es el caso. Para este caso, Reynolds presenta en [Rey98, página 230] la siguiente regla para el caso de la definición de una sola variable:

$$\frac{\rho \vdash (\lambda u\,.\,e)(\lambda u\,.\,\text{letrec } v = \lambda u\,.\,e' \text{ in } e') \Rightarrow \alpha}{\rho \vdash \text{letrec } v = \lambda u\,.\,e' \text{ in } e \Rightarrow \alpha}$$

La idea intuitiva es que en la premisa se ha *desplegado una vez* la definición recursiva. Al tener llamada por valor, el argumento $(\lambda u \,.\, \text{letrec } v = \lambda u \,.\, e' \text{ in } e')$ solo se puede evaluar si es una función, que se evalúa a una clausura.

Hemos generalizado esta regla al caso de Mini-ML, donde tenemos definiciones de patrones:

$$\frac{\rho \vdash (\lambda \text{P} \,.\, \text{E}')\ \text{E}^* \Rightarrow \alpha}{\rho \vdash \text{letrec P} = \text{E in E}' \Rightarrow \alpha}$$

donde P y E tienen la misma forma (en lo que se refiere al anidamiento), E solo contiene λ-abstracciones, y E^* tiene la misma forma que E pero donde el cuerpo de cada función se ha sustituido por un letrec, como hemos hecho en el caso anterior. La implementación en Maude, que viene a ser equivalente a la correspondiente definición formal de E^*, es la siguiente:

```
op e* : Pat Exp Exp -> Exp .

eq e*(P, E, \ P' . E1) = \ P' . (letrec P = E in E1) .
eq e*(P, E, ( E1, E2 )) = ( e*(P, E, E1), e*(P, E, E2) ) .

crl [letrec] : RO |- letrec P = E in E' => A
            if RO |- (\ P . E') e*(P, E, E) => A .
```

Ahora podemos evaluar otro de los ejemplos de [Kah87],

$$\begin{aligned} \text{letrec } (even, odd) \quad &= \quad (\lambda x \,.\, \text{if } x = 0 \text{ then true else } odd(x-1), \\ &\qquad\quad\ \ \lambda x \,.\, \text{if } x = 0 \text{ then false else } even(x-1)) \\ \text{in } even(3). & \end{aligned}$$

```
Maude> rew
    nil |- letrec (id('even), id('odd)) =
               ( (\ id('x) . (if (id('x) = 0) then true
                              else (id('odd) (id('x) - s(0))))),
                 (\ id('x) . (if (id('x) = 0) then false
                              else (id('even) (id('x) - s(0)))))
               )
           in id('even) s(s(s(0))) .

result TruthVal: false
```

6.4. Conclusiones

En este capítulo hemos mostrado cómo el enfoque de transiciones como reescrituras, que habíamos usado en el capítulo anterior para el cálculo CCS, puede utilizarse también para implementar de forma sencilla gran variedad de semánticas operacionales de lenguajes de programación. Hemos visto semánticas de evaluación y computación (incluyendo una

basada en una máquina abstracta) para lenguajes funcionales, con llamada por valor y por nombre; y semánticas de evaluación y computación para lenguajes imperativos, incluyendo no determinismo. De varias de ellas hemos visto además diferentes variantes.

En ocasiones ha sido necesario concretar algún detalle de la definición matemática de alguna semántica, como cuando en esta aparecían puntos suspensivos en alguna de las reglas semánticas. La potencia de Maude a la hora de definir los operadores sintácticos, incluyendo las propiedades de asociatividad y elemento neutro, y el encaje de patrones módulo estas propiedades, nos han permitido concretar estos detalles de forma clara y sencilla, resolviendo, por ejemplo, la elección no determinista de uno de los argumentos de una llamada a función para ser reducido. También hemos podido definir al mismo nivel el operador de sustitución sintáctica, utilizado en varias de las definiciones semánticas, incluyendo la generación de variables *nuevas* (no utilizadas en la expresión objeto de evaluación) para evitar la captura de variables libres.

En el ejemplo del lenguaje Mini-ML hemos encontrado una regla semántica (la del operador letrec) que no podía ser implementada directamente. El problema radicaba en que la semántica no era tan *operacional* como se pretendía, ya que llegado un cierto momento hacía falta adivinar el valor que se quería calcular para poder inferirlo. Cierto es que estas reglas semánticas no suelen ser habituales, por ser su significado bastante poco intuitivo. La solución ha sido encontrar una regla semántica alternativa, esta sí implementable.

También hemos mostrado cómo se puede *trazar* en Maude la aplicación de las reglas de reescritura, lo que puede ayudar a entender la semántica y cómo sus reglas se aplican, y en qué orden, para demostrar un juicio.

Capítulo 7

Una herramienta para Full LOTOS

La técnica de descripción formal Full LOTOS [ISO89] fue desarrollada en el marco de ISO para afrontar el problema de la especificación formal de sistemas distribuidos abiertos. La parte de descripción del comportamiento de procesos LOTOS (conocida como Basic LOTOS) está basada en álgebras de procesos, tomando prestadas ideas de CCS [Mil89] y CSP [Hoa85], y el mecanismo para definir y tratar tipos de datos se basa en ACT ONE [EM85].[1] En 1989 LOTOS se convirtió en un estándar internacional (IS-8807), y desde su estandarización ha sido utilizado para describir cientos de sistemas. La mayor parte de su éxito se debe a la existencia de buenas herramientas donde las especificaciones se pueden ejecutar, comparar y analizar.

El estándar define la semántica de LOTOS mediante un sistema de transiciones etiquetadas, donde cada variable de datos es instanciada con todos los posibles valores, lo que puede dar lugar a que un estado tenga infinitas transiciones, tanto en anchura como en profundidad. Esta es la razón por la cual muchas de las herramientas ignoran o restringen el uso de los tipos de datos. Calder y Shankland [CS00, CS01] han definido una semántica *simbólica* para LOTOS que da significado a procesos simbólicos o parametrizados por los datos (véase Sección 7.2) evitando la ramificación infinita.

En este capítulo se describe una herramienta formal basada en dicha semántica simbólica, por medio de la cual se pueden ejecutar especificaciones sin restricciones sobre los tipos de datos utilizados. Las características reflexivas de la lógica de reescritura y las propiedades de Maude como metalenguaje hacen posible que podamos implementar la herramienta completa en el mismo marco semántico. Así, hemos obtenido una implementación de la semántica operacional de la parte de procesos de LOTOS, que se ha integrado con las especificaciones en ACT ONE, y hemos construido un entorno para tal herramienta, que incluye el tratamiento de la entrada y salida de especificaciones LOTOS cualesquiera. Además, nuestro objetivo ha sido el de obtener una herramienta que pueda ser utilizada por un usuario cualquiera, sin conocimientos de la implementación concreta, y donde la representación de la semántica se hace a un alto nivel, para que pueda ser comprendida y modificada por cualquiera que tenga conocimientos de semánticas operacionales.

[1] La unión de las partes de descripción del comportamiento de procesos y de los tipos de datos se conoce como Full LOTOS. Nosotros utilizaremos aquí el término LOTOS para referirnos al lenguaje completo.

El enfoque seguido en este capítulo para implementar la semántica simbólica de LOTOS es el de transiciones como reescrituras utilizado en el Capítulo 5 para la semántica de CCS. La semántica de LOTOS es bastante más compleja, y la utilización de especificaciones de tipos abstractos de datos definidos por el usuario, ha presentado nuevos problemas, cuyas soluciones presentaremos a continuación.

El enfoque alternativo de representación de reglas de inferencia como reescrituras también ha sido utilizado para implementar la semántica simbólica de LOTOS, construyendo una herramienta similar a la presentada en este capítulo, donde la semántica también se ha integrado con especificaciones ACT ONE, aunque siguiendo soluciones diferentes a algunos de los problemas encontrados. La herramienta se ha integrado con Full Maude. Esta implementación alternativa se ha presentado en el informe técnico [Ver02a]. En la Sección 7.7 comentaremos los aspectos novedosos de esta implementación respecto al caso de CCS, y compararemos las dos implementaciones de la semántica de LOTOS.

El trabajo descrito en este capítulo ha sido publicado en el informe técnico [Ver02b] y ha sido presentado en la conferencia *Formal Techniques for Networked and Distributed Systems, FORTE 2002*, dando lugar a la publicación [Ver02c]. Este trabajo se engloba dentro del proyecto EPSRC británico *Developing Implementation and Extending Theory: A Symbolic Approach to Reasoning about LOTOS* [SC02]. Una panorámica de todo el trabajo realizado relacionado con este proyecto se ha publicado en [SBM+02].

7.1. LOTOS

En esta sección presentamos una visión general breve de la sintaxis del lenguaje LOTOS, y una idea intuitiva de su semántica, adaptando una descripción presentada en [CS00]. Más detalles sobre LOTOS pueden encontrarse en [ISO89, BB87].

Componentes sintácticas

expresiones de comportamiento	denotadas por P, Q
predicados de selección	(expresiones booleanas) denotados por SP
puertas	en su forma más simple se denotan por g, h. Llamamos G al conjunto de nombres de puertas, y a recorre $G \cup \{\delta, \mathbf{i}\}$. (Las acciones especiales δ y $\mathbf{i}$ se explicarán más adelante.)
comunicaciones	(los datos asociados con eventos) denotados por d y tienen la forma $!E$ o $?x{:}S$. El conjunto de eventos *estructurados* (puerta más datos) se denota por Act y α recorre $Act \cup \{\delta, \mathbf{i}\}$. Nota: $G \subseteq Act$.
expresiones de datos (abiertas)	denotadas por E
términos de datos cerrados	denotados por v
parámetros de salida	denotados por ep
variables	denotadas por x, y, z
tipos de datos	denotados por S

Operadores LOTOS

Como en CCS y CSP, las componentes principales de LOTOS son las acciones y los procesos. Los procesos básicos en LOTOS son: **stop**, el proceso parado que no puede realizar ninguna acción, y **exit**$(ep_1, \ldots, ep_n)$, el proceso que representa la terminación con éxito que devuelve los valores indicados. También hay un proceso **exit** sin devolución de valores.

Las acciones ocurren en puertas, y pueden tener o no asociadas comunicaciones de datos. Por ejemplo, $g\ d_1 \ldots d_n$ es una acción que ocurre en la puerta g, con comunicación de los datos $d_1 \ldots d_n$. Una comunicación de datos puede ser un valor (denotado por `!`, por ejemplo, g`!`4) o una variable sobre un conjunto de valores (denotado por `?`, por ejemplo, g`?`x`:`Nat). Las acciones pueden estar sujetas a *predicados de selección*, $g\ d_1 \ldots d_n$`[`SP`]`, donde SP es una condición booleana que restringe el conjunto de valores permitidos. Hay dos acciones especiales: **i**, que representa un evento interno, no observable (τ en CCS), y δ, que representa el evento de terminación con éxito (y puede tener datos asociados).

Las acciones y los procesos se combinan utilizando los operadores siguientes:

prefijo de acción	$g\ d_1 \ldots d_n$`;`P, representa el proceso que primero realiza la acción $g\ d_1 \ldots d_n$ y después se comporta como P.
elección	P_1`[]`P_2, representa el proceso que puede comportarse como P_1 o como P_2. La elección es no determinista si la acción inicial de cada rama es la misma. Hay otras dos versiones especializadas de elección: una elección no determinista de un nombre de puerta de una lista, **choice** g **in** `[`g_1`,`$\ldots$`,`g_n`]` `[]` P; y una elección no determinista de un valor de un tipo dado, **choice** $x : S$ `[]` P.
paralelismo	P_1 `|[`g_1`, ..., `g_n`]|` P_2, representa la evolución en paralelo de los procesos P_1 y P_2, sincronizando en las acciones sobre las puertas de la lista $g_1, \ldots, g_n$. En consecuencia una acción en $g_1, \ldots, g_n$ no puede ocurrir a menos que ambos procesos estén dispuestos a realizarla. Las acciones sobre las otras puertas pueden ocurrir separadamente en uno y otro proceso. Hay tres casos especiales: sin sincronización en ninguna puerta o *entrelazado* (P_1 `|||` P_2); sincronización en todas las puertas de G (P_1 `||` P_2); y **par** g **in** `[`g_1`,`$\ldots$`,`g_n`]` op P, donde op es uno de los tres operadores de paralelo ya vistos, que representa la instanciación de n copias de P combinadas en paralelo con op, reemplazando g por g_i en la i-ésima instancia de P.
habilitación	P_1 `>>` P_2, representa el proceso que se comporta como P_1 y cuando este termina con éxito, pasa a comportarse como P_2. La habilitación también puede estar parametrizada con datos: P_1 `>>` **accept** $x_1 : S_1$`, ...,` $x_n : S_n$ **in** P_2, hace que los valores se pasen desde el **exit** de terminación de P_1 ligándose en P_2 a las variables $x_1, \ldots, x_n$.

deshabilitación	P_1 `[>` P_2, representa el proceso que se comporta como P_1 pero que en cualquier momento puede ser interrumpido por P_2 que toma el control, que nunca vuelve a P_1. Si P_1 termina con éxito, P_2 ya no puede interrumpir.
guarda	`[`SP`]` `->` P, representa el proceso que se comporta como P si se cumple la condicion SP. En caso contrario, se comporta como **stop**.
ocultamiento	**hide** $g_1, \ldots, g_n$ **in** P, representa el proceso que se comporta como P, pero cuando realiza una acción en $g_1, \ldots, g_n$, esta se convierte en la acción interna **i**, de forma que no puede ser observada por el entorno. Se utiliza principalmente para forzar a los procesos dentro de P a que se comuniquen solo entre ellos.
declaración de variables	**let** x_1 `=` E_1 `,` $\ldots$ `,` x_n `=` E_n **in** P, representa el proceso P, donde la variable x_i se ha ligado al valor de E_i.

La recursión también está permitida, y se realiza a través de la instanciación de procesos definidos.

7.2. Semántica simbólica para LOTOS

La implementación de la semántica simbólica de LOTOS que presentamos aquí se basa en el trabajo presentado en [CS00, CS01]. La semántica simbólica de LOTOS asocia un sistema simbólico de transiciones a cada expresión de comportamiento P. Siguiendo las ideas presentadas en [HL95], Calder y Shankland definen los *sistemas simbólicos de transiciones* (STS) como sistemas de transiciones que separan la información sobre los datos del comportamiento de los procesos, considerando los datos como simbólicos. Los STS son sistemas de transiciones etiquetadas que contienen variables, tanto en los estados como en las transiciones, y condiciones que determinan la validez de la transición.

Un *sistema de transiciones simbólico* esta formado por:

- Un conjunto (no vacío) de estados. Cada estado T tiene asociado un conjunto de variables libres, denotado por $fv(T)$.
- Un estado inicial diferenciado, T_0.
- Un conjunto de transiciones del tipo $T \xrightarrow{b \quad \alpha} T'$, donde α es un evento simple o estructurado y b es una expresión booleana tal que $fv(T') \subseteq fv(T) \cup fv(\alpha)$ y $fv(b) \subseteq fv(T) \cup fv(\alpha)$.

En [CS01] se explica la intuición detrás de la definición y las características principales de esta semántica, junto con los axiomas y reglas de inferencia para cada operador LOTOS. Nosotros veremos estas reglas, junto con su representación en Maude, en la Sección 7.3.2.

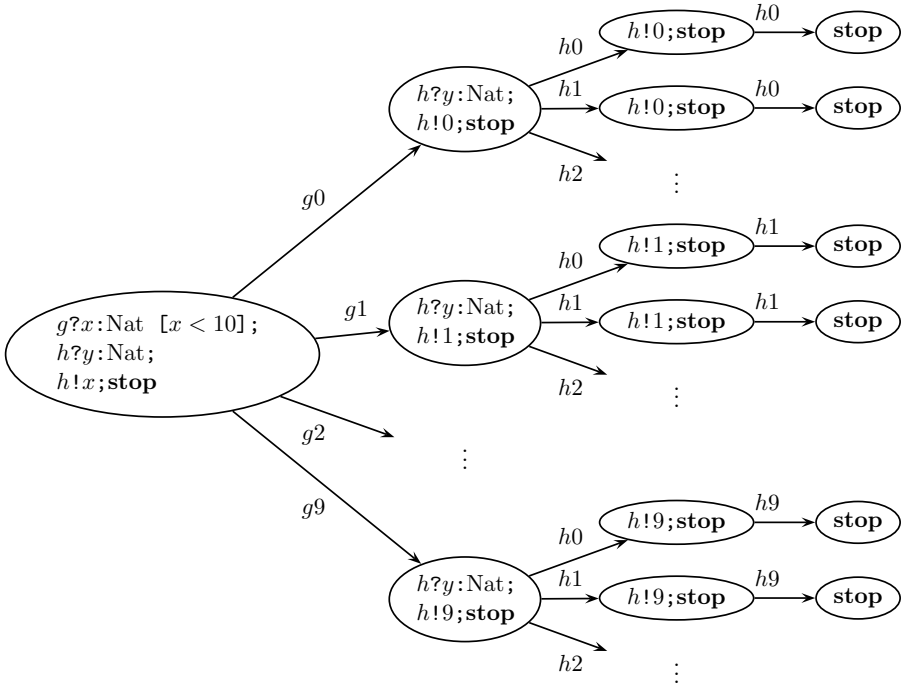


Figura 7.1: Semántica estándar: Sistema de transiciones etiquetadas.

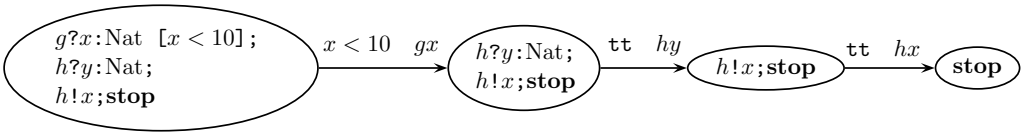


Figura 7.2: Semántica simbólica: Sistema simbólico de transiciones.

En las Figuras 7.1 y 7.2 se presenta un ejemplo que ilustra las diferencias entre la semántica estándar y la semántica simbólica. Los nodos están etiquetados por la correspondiente expresión de comportamiento en LOTOS.

En la semántica estándar, las comunicaciones de salida se instancian con datos concretos. Por tanto, en la Figura 7.1, las comunicaciones de tipo ? corresponden a muchas, o a un número infinito de transiciones, cada una etiquetada con la comunicación de un dato concreto. Incluso en este pequeño ejemplo, está claro que pueden aparecer sistemas de estados infinitos aun a partir de procesos simples. El problema empeora con el uso del paralelismo.

En la semántica simbólica, los estados pueden estar etiquetados por expresiones de comportamiento *abiertas* conteniendo variables, como por ejemplo $h\,!\,x$ **; stop**, y las transiciones ofrecen variables, bajo ciertas condiciones. Estas condiciones determinan el conjunto de valores que pueden ser utilizados para sustituir a las variables. Mientras que el sistema de la Figura 7.1 es de ramificación infinita, el sistema de la Figura 7.2 tiene solo ramificación finita.

La semántica simbólica no es suficiente por sí sola para describir cualquier comportamiento LOTOS. Si el comportamiento cicla, entonces hay que introducir un mecanismo de sustitución que permita que las variables tengan un valor diferente en cada iteración. Esto se consigue en [CS01] utilizando el concepto de *términos*. Un término está formado por un sistema de transiciones simbólico y una sustitución asociada. La sustitución se aplica paso a paso, según se va recorriendo el sistema de transiciones (véase la Sección 7.3.3).

En las próximas secciones no se incluirá el código Maude completo necesario para la implementación de la herramienta descrita, sino solo código ilustrativo de todas las ideas utilizadas y presentadas. El código completo puede encontrarse, junto con explicaciones, en el informe técnico [Ver02b], y el código Maude ejecutable y ejemplos de especificaciones en la página web http://dalila.sip.ucm.es/~alberto/tesis.

7.3. Semántica simbólica de LOTOS en Maude

Para implementar la semántica simbólica de LOTOS en Maude seguimos el enfoque de transiciones como reescrituras. Así, interpretamos la transición de LOTOS $T \xrightarrow{b \quad \alpha} T'$ como la reescritura $T \longrightarrow \{b\}\{\alpha\}T'$.

7.3.1. Sintaxis de LOTOS

Existen dos sintaxis diferenciadas: la sintaxis concreta utilizada por quien escribe especificaciones (véase Sección 7.5), y la sintaxis abstracta utilizada por la implementación de la semántica, que es la introducida en esta sección. Se define en el módulo funcional de Maude `LOTOS-SYNTAX`, el cual incluye el módulo `DATAEXP-SYNTAX`. Utilizamos los identificadores con comilla predefinidos para construir identificadores LOTOS para variables, tipos, puertas y procesos. Los booleanos son el único tipo predefinido. La sintaxis LOTOS se extiende con una sintaxis definida por el usuario cuando se utilizan especificaciones de tipos de datos ACT ONE. Los valores de estos tipos de datos extenderán el tipo `DataExp`. En la Sección 7.5 veremos cómo se produce esta extensión.

```
fmod DATAEXP-SYNTAX is
  protecting QID .

  sort VarId .
  op V : Qid -> VarId .

  sort DataExp .
  subsort VarId < DataExp . *** A LOTOS variable is a data expression.
  subsort Bool < DataExp .  *** Booleans are a predefined data type.
  subsort VarId < Bool .

  sort DataExpList .
  subsort DataExp < DataExpList .
  op nilDEL : -> DataExpList .
  op _,_ : DataExpList DataExpList -> DataExpList [assoc prec 35] .
endfm
```

```
fmod LOTOS-SYNTAX is
  protecting DATAEXP-SYNTAX .

  *** identifiers contructors
  sorts SortId GateId ProcId .

  op S : Qid -> SortId .
  op G : Qid -> GateId .
  op P : Qid -> ProcId .
```

Presentamos a continuación los procesos básicos de LOTOS: el proceso que no hace nada, y el proceso que termina con éxito, posiblemente devolviendo valores. Para simplificar, y como se hace en [CS01], supondremos que solo puede aparecer un parámetro en una expresión de comportamiento `exit`. Dicho parámetro puede ser un valor o una variable sobre un conjunto de valores.

```
  sort BehaviourExp .

  op stop : -> BehaviourExp [format (b! o)] .
  op exit : -> BehaviourExp .
  op exit(_) : ExitParam -> BehaviourExp .

  sort ExitParam .
  subsort DataExp < ExitParam .

  op any_ : SortId -> ExitParam .
```

Las acciones ocurren a través de puertas, y pueden estar acompañadas o no de comunicación de datos. Nosotros hemos preferido eliminar la restricción incluida en [CS01] por la cual solo se podía ofrecer un dato a través de una acción, permitiendo comunicaciones múltiples. Las acciones pueden estar sujetas a *predicados de selección*, los cuales son condiciones booleanas que pueden restringir los valores comunicados. La acción especial `i` representa un evento interno (no observable).

```
  sorts SimpleAction StrucAction Action Offer SelecPred IdDecl .

  subsort GateId < SimpleAction .
  subsorts SimpleAction StrucAction < Action .

  op i : -> SimpleAction .
  op __ : GateId Offer -> StrucAction [prec 30 gather(e e)] .

  op !_ : DataExp -> Offer [prec 25] .
  op ?_ : IdDecl -> Offer [prec 25] .
  op _:_ : VarId SortId -> IdDecl [prec 20] .

  op __ : Offer Offer -> Offer [assoc prec 27] .
```

```
op _[_] : SimpleAction SelecPred -> Action [prec 30] .
op _[_] : StrucAction SelecPred -> Action [prec 30] .

subsort Bool < SelecPred .
```

Enumeramos a continuación los operadores restantes junto con diversos operadores auxiliares. Obsérvese que los operadores que definen expresiones de comportamiento se han declarado con el atributo `frozen`. La razón es la misma que explicábamos en la Sección 5.2: evitar reescrituras de subtérminos de una expresión de comportamiento que darían lugar a términos mal formados.

```
*** action prefixing
op _;_ : Action BehaviourExp -> BehaviourExp [frozen prec 35] .

*** choice
op _[]_ : BehaviourExp BehaviourExp -> BehaviourExp [assoc frozen prec 40] .
op choice_[]_ : IdDecl BehaviourExp -> BehaviourExp [frozen prec 40] .
op choice_in[_][]_ : GateId GateIdList BehaviourExp -> BehaviourExp
                     [frozen prec 40] .

*** parallelism
sort GateIdList .
subsort GateId < GateIdList .
op nilGIL : -> GateIdList .
op ALL : -> GateIdList .
op _,_ : GateIdList GateIdList -> GateIdList [assoc prec 35] .

eq nilGIL, GIL:GateIdList =  GIL:GateIdList .
eq GIL:GateIdList, nilGIL =  GIL:GateIdList .
eq g:GateId, ALL = ALL .
eq ALL, g:GateId = ALL .

sort ParOp .
op |[_]| : GateIdList -> ParOp .
op || : -> ParOp .
op ||| : -> ParOp .
op ___ : BehaviourExp ParOp BehaviourExp -> BehaviourExp [frozen prec 40] .
op par_in[_]__ : GateId GateIdList ParOp BehaviourExp ->
                 BehaviourExp [frozen prec 40] .

*** enable
op _>>_ : BehaviourExp BehaviourExp -> BehaviourExp [frozen prec 40] .
op _>> accept_in_ : BehaviourExp IdDecl BehaviourExp -> BehaviourExp
                    [frozen prec 40] .

*** disable
op _[>_ : BehaviourExp BehaviourExp -> BehaviourExp [frozen prec 40] .

*** guard
op [_]->_ : SelecPred BehaviourExp -> BehaviourExp [frozen prec 40] .
```

```
  *** hide
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [frozen prec 40] .

  *** variable declaration
  op let_=_in_ : VarId DataExp BehaviourExp -> BehaviourExp [prec 40] .
```

En la sintaxis abstracta toda instanciación de proceso contiene una lista de identificadores de puertas y una lista de expresiones de datos, si bien ambas pueden ser vacías.

```
  *** process instantiation
  op _[_](_) : ProcId GateIdList DataExpList -> BehaviourExp [frozen prec 30] .

endfm
```

7.3.2. Semántica simbólica de LOTOS

Primero introducimos los contextos, de tipo `Context`, utilizados para mantener las definiciones de procesos introducidos en la especificación LOTOS. Para poder ejecutar una instanciación de proceso, se tiene que buscar en el contexto su correspondiente definición de proceso. El contexto actual se construye cuando la especificación LOTOS es introducida en la herramienta. Veremos cómo hacerlo en la Sección 7.6.4. En la semántica, se parte de una constante `context` que representa la colección de definiciones de procesos. Más formalmente, podríamos decir que la semántica está parametrizada por esta constante, que será instanciada cuando se utilice una especificación LOTOS concreta.

```
fmod CONTEXT is
  protecting LOTOS-SYNTAX .

  sorts ProcDef Context .
  subsorts ProcDef < Context .

  op process : ProcId GateIdList DataExpList BehaviourExp -> ProcDef .

  op nil : -> Context .
  op _&_ : [Context] [Context] -> [Context] [assoc comm id: nil prec 42] .

  op _definedIn_ : ProcId Context -> Bool .
  op def : Context ProcId -> [ProcDef] .

  op context : -> Context .

  vars X X' : ProcId .
  var P : BehaviourExp .
  var GIL : GateIdList .
  var DEL : DataExpList .
  var C : Context .
```

Un contexto está bien definido si cada identificador de proceso se define como mucho una vez. Utilizamos un axioma condicional de pertenencia (`cmb`) para establecer qué términos de la familia `[Context]` son contextos bien definidos (de tipo `Context`).

```
  cmb process(X,GIL,DEL,P) & C : Context if not(X definedIn C) .

  eq X definedIn nil = false .
  eq X definedIn (process(X',GIL,DEL,P) & C) = (X == X') or (X definedIn C) .
```

La evaluación de `def(C,X)` devuelve la definición de proceso asociada al identificador de proceso `X` si esta existe.

```
  eq  def(process(X,GIL,DEL,P) & C, X) = process(X,GIL,DEL,P) .
  ceq def(process(X',GIL,DEL,P) & C, X) = def(C, X) if X =/= X' .
endfm
```

Ahora ya podemos implementar la semántica simbólica de LOTOS. Primero definimos los elementos de las transiciones simbólicas, es decir, los eventos y las condiciones de las transiciones.

```
mod LOTOS-SEMANTICS is
  protecting CONTEXT .
  protecting ORACLE .

  sorts SimpleEv StructEv EOffer Event TransCond .
  subsort SimpleAction < SimpleEv .
  subsorts SimpleEv StructEv < Event .
  subsort DataExp < EOffer .

  op delta : -> GateId .
  op __ : GateId EOffer -> StructEv [prec 30 gather(e e)] .

  op __ : EOffer EOffer -> EOffer [assoc prec 27] .

  *** Trace of events (used for the tool output)

  sort Trace .
  subsort Event < Trace .
  op nil : -> Trace .
  op __ : Trace Trace -> Trace [assoc id: nil prec 35 gather(e E)] .

  *** transition conditions

  subsort Bool < TransCond .
  op _=_ : DataExp DataExp -> TransCond [prec 25] .
  op _/\_ : TransCond TransCond -> TransCond [assoc comm] .

  vars B B' : Bool .
  var A : SimpleAction .
```

```
vars E E' E1 E2 : DataExp .
vars g g' g'' gi : GateId .
vars O O' : Offer .
vars EO EO1 EO2 : EOffer .
var SP : SelecPred .
vars x y z : VarId .
var S : SortId .
var p : ProcId .
vars GIL GIL' : GateIdList .
var OP : ParOp .
vars DEL DEL' : DataExpList .
vars b b' b1 b2 : TransCond .
vars a a' a1 a2 : Event .
vars P P' P1 P2 P1' P2' : BehaviourExp .
var N : MachineInt .

eq true /\ b = b .
```

A continuación, definimos un operador para construir transiciones simbólicas. Una transición $T \xrightarrow{b \quad \alpha} T'$ será representada como una reescritura $T \longrightarrow \{b\}\{\alpha\}T'$, donde el término de la parte derecha es del tipo `TCondEventBehExp`.

```
sort TCondEventBehExp .
subsort BehaviourExp < TCondEventBehExp .
op {_}{_}_ : TransCond Event TCondEventBehExp -> TCondEventBehExp [frozen] .
```

Igual que en la Sección 5.2, las reglas de reescritura que manejaremos tienen la propiedad de "incrementar el tipo", ya que reescriben términos de tipo `BehaviourExp` a términos de su supertipo `TCondEventBehExp`. Esto evita la aparición de términos mal formados en los que los subtérminos de un operador LOTOS no sean expresiones de comportamiento.

Además, el operador `{_}{_}_` utilizado para construir los valores a la derecha de las reglas de reescritura que representen las reglas semánticas se ha declarado como `frozen`. La razón es evitar que se realicen reescrituras innecesarias que podrían dar lugar a procesos infinitos de búsqueda. De esta forma, con el comando `search` o la función `metaSearch` al metanivel, podremos conseguir los sucesores en un paso de una expresión de comportamiento (véanse las Secciones 7.3.4 y 7.4).

Antes de definir las reglas de la semántica, definimos varias funciones utilizadas por esta. En la semántica se utiliza un conjunto **new-var** de nombres de variables nuevas, no utilizadas. Como se dice en [CS01], hablando estrictamente, cualquier referencia a este conjunto requiere un contexto, que incluirá los nombres de variables que ya han aparecido[2]. En vez de complicar la implementación de la semántica con este nuevo contexto, hemos preferido utilizar una utilidad predefinida de Maude, importada del módulo `ORACLE`[3], don-

[2] Nótese que *no* estamos en la misma situación que teníamos en la Sección 6.1.1, cuando para definir el operador de sustitución $e[e'/x]$ necesitábamos una nueva variable que no apareciera ni en e ni en e'. Aquí necesitamos una variable que no haya sido utilizada todavía.

[3] La presencia de este módulo puede cambiar en futuras implementaciones del sistema Maude 2.0.

de se define una constante NEWQ. Cada vez que NEWQ se reescribe, lo hace a un identificador con comilla distinto. Esto nos sirve para definir variables nuevas de la siguiente manera:

```
op new-var : -> VarId .
eq new-var = V(NEWQ) .
```

La sustitución (de datos) de x por e se denota por $[e/x]$. Parecería que esta operación debiera ser fácil de implementar ecuacionalmente. De hecho más abajo presentamos algunas ecuaciones de ejemplo, que muestran cómo la operación de sustitución se distribuye sobre la sintaxis de las expresiones de comportamiento. La implementación completa (o tan completa como se puede conseguir en este momento) puede encontrarse en [Ver02b]. Sin embargo, si permitimos expresiones de datos con sintaxis definida por el usuario, por medio de especificaciones ACT ONE, tenemos que en este punto no podemos definir completamente la operación, al no conocerse todavía la sintaxis de las expresiones de datos. En la Sección 7.5.1 describiremos cómo el módulo que contiene la nueva sintaxis de datos se extiende de forma automática para definir esta operación sobre las nuevas expresiones de datos.

```
sort Substitution .
op `[`] : -> Substitution .
op `[_/_`] : DataExp DataExp -> Substitution .
op __ : Substitution Substitution -> Substitution .

op __ : DataExp Substitution -> DataExp .
op __ : Offer Substitution -> Offer .
op __ : BehaviourExp Substitution -> BehaviourExp .

eq P [] = P .

eq ! E1 [E' / x ] = ! (E1 [E' / x ]) .
eq ? y : S [E' / x ] = ? y : S .
eq ! E1 O [E' / x ] = ! (E1 [E' / x ]) (O[E' / x ]) .
eq ? y : S O [E' / x ] = ? y : S (O[E' / x ]) .

eq A [E' / x ] = A .
eq g'' O [E' / x ] = g''(O[E' / x ]) .

eq stop [E' / x ] = stop .
eq A:Action ; P [E' / x ] = (A:Action[E' / x ]) ; (P[E' / x ]) .
eq P1 [] P2 [E' / x ] = (P1[E' / x ]) [] (P2[E' / x ]) .
eq P1 OP P2 [E' / x ] = (P1[E' / x ]) OP (P2[E' / x ]) .
eq par g in [GIL] OP P [E' / x ] = par g in [GIL] OP (P[E' / x ]) .
eq P1 >> P2 [E' / x ] = (P1[E' / x ]) >> (P2[E' / x ]) .
```

También declaramos una operación de sustitución de identificadores de puertas en una expresión de comportamiento, definida de forma similar.

```
op `[_/_`] : GateId GateId -> Substitution .
```

La función **name** : Act $\cup$ $\{\delta, \mathbf{i}\} \rightarrow G \cup \{\delta, \mathbf{i}\}$ extrae el nombre de la puerta de un evento estructurado:

```
op name : Event -> Event .

eq name(i) = i .
eq name(g) = g .
eq name(g EO) = g .
```

Necesitamos también un predicado que nos diga si un identificador de puerta dado aparece en una lista de identificadores de puertas.

```
op _in_ : Event GateIdList -> Bool .

eq i in GIL = false .
eq g in nilGIL = false .
eq g in ALL = true .
eq g in g' = (g == g') .
eq g in (g' , GIL) = (g == g') or (g in GIL) .
```

También se utiliza una función *vars* que nos da el conjunto de variables que aparecen en una expresión de comportamiento. Y aquí tenemos el mismo problema que teníamos con la operación de sustitución, es decir, no podemos definir completamente la operación *vars* a este nivel, ya que la sintaxis definida por el usuario para construir expresiones de datos es desconocida. En la Sección 7.5.1 veremos cómo se define de forma automática para las nuevas expresiones de datos.

```
sort VarSet .
subsort VarId < VarSet .

op mt : -> VarSet .
op _U_ : VarSet VarSet -> VarSet [assoc comm id: mt] .
eq x U x = x . *** idempotency

op _in_ : VarId VarSet -> Bool .
op _subseteq_ : VarSet VarSet -> Bool .
op _\_ : VarSet VarSet -> VarSet .

vars VS VS' : VarSet .

eq x in mt = false .
eq x in (y U VS) = (x == y) or (x in VS) .

eq mt subseteq VS = true .
eq (x U VS) subseteq VS' = (x in VS') and (VS subseteq VS') .

eq mt \ VS' = mt .
eq (y U VS) \ VS' = if (y in VS') then VS \ VS'
                    else y U (VS \ VS') fi .
```

```
op vars : Offer -> VarSet .
op vars : BehaviourExp -> VarSet .
op vars : DataExp -> VarSet .
```

Las siguientes ecuaciones muestran cómo la operación `vars` recorre el término extrayendo sus variables.

```
eq vars(! E) = vars(E) .
eq vars(? x : S) = x .
eq vars(! E O) = vars(E) U vars(O) .
eq vars(? x : S O) = x U vars(O) .

eq vars(stop) = mt .
eq vars(A ; P) = vars(P) .
eq vars(g O ; P) = vars(O) U vars(P) .
eq vars(P1 [] P2) = vars(P1) U vars(P2) .
eq vars(choice x : S [] P) = x U vars(P) .

eq vars(x) = x .
eq vars(true) = mt .
eq vars(false) = mt .
eq vars(not(B)) = vars(B) .
eq vars(B and B') = vars(B) U vars(B') .
eq vars(B or B') = vars(B) U vars(B') .
```

El conjunto de variables libres que aparecen en una expresión de comportamiento o de datos se define de la siguiente manera:

```
op fv : DataExp -> VarSet .
op fv : BehaviourExp -> VarSet .
op fv : Offer -> VarSet .
op bv : Offer -> VarSet .

eq fv(E) = vars(E) .   *** all variables are free in E

eq fv(stop) = mt .
eq fv(A ; P) = fv(P) .
eq fv(g O ; P) = fv(O) U (fv(P) \ bv(O)) .
eq fv(P1 [] P2) = fv(P1) U fv(P2) .
eq fv(choice x : S [] P) = fv(P) \ x .

eq fv(! E) = fv(E) .
eq fv(? x : S) = mt .
eq fv(! E O) = fv(E) U fv(O) .
eq fv(? x : S O) = mt U fv(O) .

eq bv(! E) = mt .
eq bv(? x : S) = x .
```

```
eq bv(! E O) = bv(O) .
eq bv(? x : S O) = x U bv(O) .
```

Ahora ya podemos representar las reglas de la semántica simbólica de LOTOS. Para cada operador de LOTOS, presentaremos las reglas semánticas y su representación como reglas de reescritura. Algunas de las reglas no coinciden exactamente con las presentadas en [CS00], ya que aquí se han generalizado para considerar el caso en el que se comunica más de un valor por una puerta.

axiomas de prefijo

$$a\,;P \xrightarrow{\texttt{tt} \quad a} P$$

$$g\ d_1 \ldots d_n\,;P \xrightarrow{\texttt{tt} \quad gE'_1 \ldots E'_n} P$$

$$g\ d_1 \ldots d_n\,\texttt{[}SP\texttt{]}\,;P \xrightarrow{SP \quad gE'_1 \ldots E'_n} P$$

donde $E'_i = \begin{cases} E_i & \text{si} \quad d_i = \texttt{!}E_i \\ x_i & \text{si} \quad d_i = \texttt{?}x_i\texttt{:}S_i \end{cases}$

```
*** prefix axioms
rl [prefix] : A ; P => {true}{A}P .
rl [prefix] : g O ; P => {true}{g eOffer(O)}P .
rl [prefix] : g O [SP] ; P => {SP}{g eOffer(O)}P .

op eOffer : Offer -> EOffer .

eq eOffer(! E) = E .
eq eOffer(? x : S) = x .
eq eOffer(O O') = (eOffer(O) eOffer(O')) .
```

axiomas para exit

$$\textbf{exit} \xrightarrow{\texttt{tt} \quad \delta} \textbf{stop}$$

$$\textbf{exit}(ep) \xrightarrow{\texttt{tt} \quad \delta E'} \textbf{stop}$$

$$E' = \begin{cases} E & \text{si } ep = E \\ z & \text{si } ep = \textbf{any}\ S \quad \text{donde } z \in \textbf{new-var}. \end{cases}$$

```
*** exit axioms
rl [exit] : exit => {true}{delta}stop .
rl [exit] : exit(E) => {true}{delta E}stop .
rl [exit] : exit(any S) => {true}{delta new-var}stop .
```

Obsérvese la utilización del término `new-var` en la última regla `exit`.

regla del let

$$\frac{P[E/x] \xrightarrow{b \quad \alpha} P'}{\textbf{let } x \texttt{ = } E \textbf{ in } P \xrightarrow{b \quad \alpha} P'}$$

```
*** let rule
crl [let] : let x = E in P => {b}{a}P'
         if P [E / x] => {b}{a}P' .
```

reglas de la elección sobre valores

$$\frac{P[g_i/g] \xrightarrow{b \quad \alpha} P'}{\textbf{choice } g \textbf{ in } [g_1, \ldots, g_n] \texttt{ []} \; P \xrightarrow{b \quad \alpha} P'}$$

por cada $g_i \in \{g_1, \ldots, g_n\}$

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{choice } x : S \texttt{ []} \; P \xrightarrow{b \quad \alpha} P'}$$

```
*** choice range rules
crl [choicer] : choice g in [GIL][] P => {b}{a}P'
             if select(GIL) => gi /\
                P[gi / g] => {b}{a}P' .

crl [choicer] : choice x : S [] P => {b}{a}P'
             if P => {b}{a}P' .

sort ND-GateId .
subsort GateId < ND-GateId .
op select : GateIdList -> ND-GateId .

rl [sel] : select(g) => g .
rl [sel] : select(g, GIL) => g .
rl [sel] : select(g, GIL) => select(GIL) .
```

Obsérvese cómo se utiliza un tipo `ND-GateId` y una operación `select` para representar un identificador de puerta elegido de forma no determinista a partir de una lista de identificadores. Esta elección no determinista se implementa por medio de las reglas de reescritura `sel` anteriores.

regla del paralelo con instanciación

$$\frac{P[g_1/g] \; op \; \ldots \; op \; P[g_n/g] \xrightarrow{b \quad \alpha} P'}{\textbf{par } g \textbf{ in } [g_1, \ldots, g_n] \; op \; P \xrightarrow{b \quad \alpha} P'}$$

donde *op* es uno de los operadores paralelo, `||`, `|||`, o `|[`$h_1, \ldots, h_m$`]|`.

```
*** par rule
crl [pari] : par g in [GIL] OP P => {b}{a}P'
          if unfold(g, GIL, OP, P) => {b}{a}P' .

op unfold : GateId GateIdList ParOp BehaviourExp -> BehaviourExp .

eq unfold(g, g', OP, P) = P[g' / g] .
eq unfold(g, (g', GIL), OP, P) = (P[g' / g]) OP unfold(g, GIL, OP, P) .
```

La operación `unfold` se utiliza para construir la expresión de comportamiento de la premisa de la regla.

reglas del ocultamiento

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{hide } g_1, \ldots, g_n \textbf{ in } P \xrightarrow{b \quad i} \textbf{hide } g_1, \ldots, g_n \textbf{ in } P'}$$

si $\textbf{name}(\alpha) \in \{g_1, \ldots, g_n\}$

$$\frac{P \xrightarrow{b \quad \alpha} P'}{\textbf{hide } g_1, \ldots, g_n \textbf{ in } P \xrightarrow{b \quad \alpha} \textbf{hide } g_1, \ldots, g_n \textbf{ in } P'}$$

si $\textbf{name}(\alpha) \notin \{g_1, \ldots, g_n\}$

```
*** hide rules
crl [hide] : hide GIL in P => {b}{i}hide GIL in P'
          if P => {b}{a}P' /\
            (name(a) in GIL) .
crl [hide] : hide GIL in P => {b}{a}hide GIL in P'
          if P => {b}{a}P' /\
             not(name(a) in GIL) .
```

reglas de la habilitación

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \gg \textbf{accept } x{:}S \textbf{ in } P_2 \xrightarrow{b \quad \alpha} P_1' \gg \textbf{accept } x{:}S \textbf{ in } P_2}$$

si $\textbf{name}(\alpha) \neq \delta$

$$\frac{P_1 \xrightarrow{b \quad \delta E} P_1'}{P_1 \gg \textbf{accept } x{:}S \textbf{ in } P_2 \xrightarrow{b \quad i} P_2[E/x]}$$

Y de forma similar para el operador >> sin datos.

```
*** accept rules
crl [accept] : P1 >> accept x : S in P2 => {b}{a}(P1' >> accept x : S in P2)
            if P1 => {b}{a}P1' /\
              (name(a) =/= delta) .
crl [accept] : P1 >> accept x : S in P2 => {b}{i}(P2 [E / x])
            if P1 => {b}{delta E}P1' .

crl [accept] : P1 >> P2 => {b}{a}(P1' >> P2)
            if P1 => {b}{a}P1' /\
              (name(a) =/= delta) .
crl [accept] : P1 >> P2 => {b}{i}P2
            if P1 => {b}{delta}P1' .
```

reglas de la deshabilitación

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \texttt{ [> } P_2 \xrightarrow{b \quad \alpha} P_1' \texttt{ [> } P_2}$$

si $\mathbf{name}(\alpha) \neq \delta$

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \texttt{ [> } P_2 \xrightarrow{b \quad \alpha} P_1'}$$

si $\mathbf{name}(\alpha) = \delta$

$$\frac{P_2 \xrightarrow{b \quad \alpha} P_2'}{P_1 \texttt{ [> } P_2 \xrightarrow{b \quad \alpha} P_2'}$$

```
*** disable rules
crl [disable] : P1 [> P2 => {b}{a}(P1' [> P2)
             if P1 => {b}{a}P1' /\
               (name(a) =/= delta) .
crl [disable] : P1 [> P2 => {b}{a}P1'
             if P1 => {b}{a}P1' /\
               (name(a) == delta) .
crl [disable] : P1 [> P2 => {b}{a}P2'
             if P2 => {b}{a}P2' .
```

reglas del paralelismo general (sin sincronización)

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \texttt{ |[} g_1, \ldots, g_n \texttt{]| } P_2 \xrightarrow{b\sigma \quad \alpha\sigma} P_1'\sigma \texttt{ |[} g_1, \ldots, g_n \texttt{]| } P_2}$$

$\mathbf{name}(\alpha) \notin \{g_1, \ldots, g_n, \delta\}$

$\sigma = \sigma_1 \ldots \sigma_n$, $\alpha = gE_1 \ldots E_n$, y

$$\sigma_i = \begin{cases} [z_i/x_i] & \text{si } E_i = x_i \text{ y } x_i \in vars(P_2) \quad \text{donde } z_i \in \mathbf{new\text{-}var}. \\ [\,] & \text{en otro caso} \end{cases}$$

De forma similar para P_2.

```
*** general parallelism rules (not synchronising)
crl [genpar] : P1 |[ GIL ]| P2 => {b subsPar(a, vars(P2))}
                                  {a subsPar(a, vars(P2))}
                                  ((P1' subsPar(a, vars(P2))) |[ GIL ]| P2)
            if P1 => {b}{a}P1' /\
               not(name(a) in (GIL, delta)) .
crl [genpar] : P1 |[ GIL ]| P2 => {b subsPar(a, vars(P1))}
                                  {a subsPar(a, vars(P1))}
                                  (P1 |[ GIL ]| (P2' subsPar(a, vars(P1))))
            if P2 => {b}{a}P2' /\
               not(name(a) in (GIL, delta)) .
```

La siguiente operación devuelve la sustitución necesaria para definir la regla semántica del operador de composición paralela, que depende de la acción dada y del conjunto de variables en el proceso que no realiza la acción.

```
op subsPar : Event VarSet -> Substitution .
op subsPar : EOffer VarSet -> Substitution .

eq subsPar(A, VS) = [] .
eq subsPar(g EO, VS) = subsPar(EO, VS) .

eq subsPar(E, VS) = if (E :: VarId) then
                       (if (E in VS) then [ new-var / E ] else [] fi)
                    else [] fi .
eq subsPar(E EO, VS) = if (E :: VarId) then
                          (if (E in VS) then [ new-var / E ] subsPar(EO, VS)
                           else subsPar(EO, VS) fi)
                       else subsPar(EO, VS) fi .
```

El operador de entrelazado `|||` es un simple caso particular:

```
eq P1 ||| P2 = P1 |[ nilGIL ]| P2 .
```

reglas del paralelismo general (con sincronización)

$$\frac{P_1 \xrightarrow{b_1 \quad g} P_1' \qquad P_2 \xrightarrow{b_2 \quad g} P_2'}{P_1 \texttt{ |[} g_1, \ldots, g_n \texttt{]| } P_2 \xrightarrow{b_1 \wedge b_2 \quad g} P_1' \texttt{ |[} g_1, \ldots, g_n \texttt{]| } P_2'}$$

donde $g \in \{g_1, \ldots, g_n, \delta\}$.

$$\frac{P_1 \xrightarrow{b_1 \quad gE_1 \ldots E_n} P_1' \qquad P_2 \xrightarrow{b_2 \quad gE_1' \ldots E_n'} P_2'}{P_1 \texttt{ |[} g_1, \ldots, g_n \texttt{]| } P_2 \xrightarrow{b_1 \wedge b_2 \wedge E_1 = E_1' \wedge \ldots \wedge E_n = E_n' \quad gE_1 \ldots E_n} P_1' \texttt{ |[} g_1, \ldots, g_n \texttt{]| } P_2'}$$

donde $g \in \{g_1, \ldots, g_n, \delta\}$.

```
*** general parallelism rules (synchronising without values)
crl [genpar] : P1 |[ GIL ]| P2 => {b1 /\ b2}{g}(P1' |[ GIL ]| P2')
            if P1 => {b1}{g}P1' /\
               P2 => {b2}{g}P2' /\
              (g in (GIL, delta)) .

*** general parallelism rules (synchronising with values)
crl [genpar] : P1 |[ GIL ]| P2 => {b1 /\ b2 /\ match(EO1,EO2)}
                                  {g EO1}
                                  (P1' |[ GIL ]| P2')
            if P1 => {b1}{g EO1}P1' /\
               P2 => {b2}{g EO2}P2' /\
              (g in (GIL, delta)) .

op match : EOffer EOffer -> TransCond .
eq match(E1, E2) = (E1 = E2) .
eq match(E1 EO1, E2 EO2) = (E1 = E2) /\ match(EO1, EO2) .
```

El operador de paralelo `||` es también un caso particular:

```
eq P1 || P2 = P1 |[ ALL ]| P2 .
```

reglas de la elección

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1 \texttt{ [] } P_2 \xrightarrow{b \quad \alpha} P_1'}$$

$$\frac{P_2 \xrightarrow{b \quad \alpha} P_2'}{P_1 \texttt{ [] } P_2 \xrightarrow{b \quad \alpha} P_2'}$$

```
*** choice rules
crl [choice] : P1 [] P2 => {b}{a}P1'
            if P1 => {b}{a}P1' .
crl [choice] : P1 [] P2 => {b}{a}P2'
            if P2 => {b}{a}P2' .
```

regla de la guarda

$$\frac{P \xrightarrow{b \quad \alpha} P'}{(\texttt{[}SP\texttt{] -> } P) \xrightarrow{b \wedge SP \quad \alpha} P'}$$

```
*** guard rule
crl [guard] : [ SP ] -> P => {b /\ SP}{a}P'
           if P => {b}{a}P' .
```

regla de la instanciación de proceso

$$\frac{P[g_1/h_1,\ldots,g_n/h_n][E_1/x_1,\ldots,E_m/x_m] \xrightarrow{b \quad \alpha} P'}{p[g_1,\ldots,g_n](E_1,\ldots,E_m) \xrightarrow{b \quad \alpha} P'}$$

donde $p[h_1,\ldots,h_n](x_1,\ldots,x_m) := P$ es una definición de proceso.

```
*** instantiation rule
crl [instantiation] : p[GIL](DEL) => {b}{a}P'
                   if (p definedIn context) /\
                      instantiate(def(context,p),GIL,DEL) => {b}{a}P' .
```

La operación `instantiate`, dada una definición de proceso y los identificadores de puertas y las expresiones de datos utilizadas en una instanciación del proceso, devuelve la expresión de comportamiento que define al proceso, donde los identificadores de puertas formales y los parámetros de datos han sido sustituidos por los actuales.

```
op instantiate : ProcDef GateIdList DataExpList -> BehaviourExp .
op instantiate : BehaviourExp GateIdList GateIdList -> BehaviourExp .
op instantiate : BehaviourExp DataExpList DataExpList -> BehaviourExp .

eq instantiate(process(p,GIL,DEL,P), GIL', DEL') =
    instantiate(instantiate(P,GIL,GIL'), DEL, DEL') .

eq instantiate(P, nilGIL, nilGIL) = P .
eq instantiate(P, g, g') = P [ g' / g ] .
eq instantiate(P, (g, GIL), (g', GIL')) = instantiate(P,GIL,GIL') [ g' / g ] .

eq instantiate(P, nilDEL, nilDEL) = P .
eq instantiate(P, E, E') = P [ E' / E ] .
eq instantiate(P, (E, DEL), (E', DEL')) = instantiate(P,DEL,DEL') [ E' / E ] .
```

Ahora ya hemos implementado todas las reglas semánticas para expresiones de comportamiento, y tenemos el siguiente resultado de conservación.

Conservación de transiciones simbólicas LOTOS: Dada una expresión de comportamiento LOTOS P, existen una condición de transición b, un evento a, y una expresión de comportamiento P' tales que

$$P \xrightarrow{b \quad a} P'$$

si y solo si `P` puede ser reescrito a `{b}{a}P'` utilizando las reglas de reescritura presentadas.

La semántica simbólica no propaga a la expresión de comportamiento resultante las ligaduras de la forma `x = E` que puedan aparecer en la condición de una transición cuando sincronizan dos comportamientos. Ello en realidad no es necesario, pues el valor de cualquier variable puede averiguarse consultando las condiciones en un recorrido del sistema de transiciones simbólico. Sin embargo, la herramienta LOTOS que definiremos más adelante sí que propagará estas ligaduras, para mostrar de forma más legible las posibles transiciones de un comportamiento. La propagación de las ligaduras en una condición de una transición se puede definir así:

```
ceq (E1 = E2) = (E2 = E1) if E2 :: VarId and not(E1 :: VarId) .

op apply-subst : TransCond BehaviourExp -> BehaviourExp .

eq apply-subst(B, P) = P .
eq apply-subst(E1 = E2, P) =
   if (E1 :: VarId) and not(E2 :: VarId) then P[E2 / E1]
   else if (E2 :: VarId) and not(E1 :: VarId) then P[E1 / E2]
        else P
        fi
   fi .
eq apply-subst(b /\ b', P) = apply-subst(b, apply-subst(b',P)) .
```

7.3.3. Semántica de términos

En [CS01] se define también el concepto de *término* (*term*): un término viene dado por un STS T y una sustitución σ, y se denota como T_σ. Así mismo se definen las transiciones entre términos, que implementaremos de la misma forma que lo hemos hecho con las transiciones entre expresiones de comportamiento. El término T_σ se representa con el par <T,σ>.

```
sorts term substitution .

op <_‘,_> : BehaviourExp substitution -> term [frozen] .

op ‘(‘) : -> substitution .
op ‘(_<-_‘) : DataExp VarId -> substitution .
op __ : substitution substitution -> substitution [assoc id: ()] .
op _‘[_/_‘] : substitution DataExp VarId -> substitution .
op remove : substitution VarId -> substitution .

var sig : substitution .

eq sig [ E / x ] = remove(sig,x) (E <- x) .

eq remove((), x) = () .
eq remove((E <- y) sig, x) = if (x == y) then sig
                             else (E <- y) remove(sig, x) fi .

op __ : TransCond substitution -> TransCond .
op __ : Event substitution -> Event .

eq b () = b .
eq b ((E <- x) sig) = b [E / x] sig .
eq a () = a .
eq a ((E <- x) sig) = a [E / x] sig .

op _<|_ : VarSet substitution -> substitution .
```

```
eq VS <| () = () .
eq VS <| ((E <- x) sig) = if (x in VS) then (E <- x) (VS <| sig)
                          else (VS <| sig) fi .

sort TCondEventTerm .
subsort term < TCondEventTerm .
op {_}{_}_ : TransCond Event TCondEventTerm -> TCondEventTerm [frozen] .
```

Las transiciones entre términos se definen en [CS01] de la siguiente manera:

$$
\begin{array}{rcl}
T \xrightarrow{b \quad a} T' & \text{implica} & T_\sigma \xrightarrow{b\sigma \quad a} T'_{\sigma'} \\
T \xrightarrow{b \quad gE} T' & \text{implica} & T_\sigma \xrightarrow{b\sigma \quad gE\sigma} T'_{\sigma'} \\
 & & \text{donde } fv(E) \subseteq fv(T) \\
T \xrightarrow{b \quad gx} T' & \text{implica} & T_\sigma \xrightarrow{b\sigma[z/x] \quad gz} T'_{\sigma'[z/x]} \\
 & & \text{donde } x \notin fv(T) \text{ y } z \notin fv(T_\sigma)
\end{array}
$$

En todos los casos $\sigma' = fv(T') \lhd \sigma$, donde la expresión $S \lhd \sigma$ denota la proyección sobre el dominio, es decir, la restricción de σ para que incluya solo los elementos del conjunto S.

```
crl [term] : < P, sig > => {b}{A} < P', fv(P') <| sig >
          if P => {b}{A}P' .
crl [term] : < P, sig > => {b sig}{(g E) sig} < P', fv(P') <| sig >
          if P => {b}{g E}P' /\
             fv(E) subseteq fv(P) .
crl [term] : < P, sig > => {b (sig [ z / x ])}
                           {g z}
                           < P', (fv(P') <| sig) [ z / x ] >
          if P => {b}{g x}P' /\
             not(x in fv(P)) /\
             z := new-var  .

endm
```

7.3.4. Ejemplo de ejecución

Utilizando el comando `search` de Maude, que busca todas las reescrituras de un término que encajan con un patrón dado, podemos encontrar todas las posibles transiciones de una expresión de comportamiento:

```
Maude> search
  G('g) ; G('h) ; stop
|[ G('g) ]|
  ( G('a) ; stop
   []
    G('g) ; stop ) => X:TCondEventBehExp .
```

```
Solution 1 (state 1)
X:TCondEventBehExp --> {true}{G('a)}G('g) ; G('h) ; stop |[G('g)]| stop

Solution 2 (state 2)
X:TCondEventBehExp --> {true}{G('g)}G('h) ; stop |[G('g)]| stop

No more solutions.

Maude> search G('h) ; stop |[G('g)]| stop => X:TCondEventBehExp .

Solution 1 (state 1)
X:TCondEventBehExp --> {true}{G('h)}stop |[G('g)]| stop

No more solutions.
```

Pero tenemos que escribir los identificadores utilizando la sintaxis abstracta (por ejemplo, `G('g)`) y no podemos utilizar expresiones de datos, aparte de los booleanos predefinidos, ya que no hemos incluido ninguna especificación ACT ONE. Estas especificaciones son definidas por el usuario como parte de una especificación Full LOTOS. En las siguientes secciones veremos cómo podemos dar semántica a las especificaciones ACT ONE, y cómo pueden integrarse con la implementación de la semántica de LOTOS presentada anteriormente.

7.4. Lógica modal FULL

Antes de describir la implementación de la herramienta, vamos a ver la implementación de parte de la lógica modal FULL basada en la semántica simbólica de LOTOS y descrita en [CMS01]. Lo haremos así porque parte de las operaciones utilizadas para su definición se utilizan también para la herramienta de simulación de procesos.

Además de las constantes, operadores binarios y operadores modales de la lógica modal de Hennessy-Milner (Sección 4.7), la lógica modal FULL incluye cuantificación universal y existencial sobre valores de datos, de forma que los operadores modales pueden expresar tanto cuantificación sobre transiciones como cuantificación sobre datos. Se exponen a continuación los nuevos operadores, y la idea intuitiva sobre su significado:

$\langle \exists y\ g \rangle$	Un valor, una transición g.
$\langle \forall y\ g \rangle$	Todos los valores, con suficientes transiciones g para cubrir todos los valores.
$[\exists y\ g]$	Un valor y varias transiciones g, más exactamente, todas las transiciones g correspondientes al valor en cuestión.
$[\forall y\ g]$	Todos los valores, todas las transiciones g.

En el artículo [CMS02] se presentan con detalle diversos ejemplos de fórmulas que manejan estos operadores modales, así como procesos que cumplen esas fórmulas. Nosotros

consideraremos aquí el siguiente proceso P,

```
process P [g, h, k] : exit :=
   g?x:Num [x < 5]; h; exit              (1)
[] g!4; k; exit                          (2)
[] g?x:Num [x = 5]; (   h; exit          (3)
                     [] k; exit )        (4)
[] g!5; h; exit                          (5)
[] g?x:Num [x > 5]; h; exit              (6)
[] g!10; k; exit                         (7)
endproc
```

donde el tipo `Num` representa los enteros del 1 al 10, y donde hemos numerado las distintas ramas que puede seguir el proceso, para utilizar estos números en las explicaciones que damos a continuación.

Veamos algunos ejemplos de fórmulas FULL que satisface el proceso `P`. Por ejemplo, el predicado "`P` *puede realizar una acción* `g` *comunicando algún valor igual a* 4", puede expresarse en FULL como $\texttt{P} \models \langle \exists y\ \texttt{g} \rangle (y = 4)$. La rama (1) muestra que `P` satisface el predicado. También se cumple que, "*para todos los valores,* `P` *puede realizar una acción* `g`*, y después una acción* `h`", $\texttt{P} \models \langle \forall y\ \texttt{g} \rangle \langle \texttt{h} \rangle \texttt{tt}$, como muestran las ramas (1), (5) y (6). Sin embargo, `P` no cumple la propiedad similar $\langle \forall y\ \texttt{g} \rangle \langle \texttt{k} \rangle \texttt{tt}$. Sí cumple que "*para algún valor, independientemente de la posibilidad de hacer la acción* `g` *que se elija, después es posible hacer una acción* `h`", $P \models [\exists y\ \texttt{g}] \langle \texttt{h} \rangle \texttt{tt}$, como muestran las ramas (3) y (5). En cambio, tampoco cumple $[\forall y\ \texttt{g}] \langle \texttt{h} \rangle \texttt{tt}$, por culpa de la rama (2). Si la fórmula se extiende a $[\forall y\ \texttt{g}](\langle \texttt{h} \rangle \texttt{tt} \vee \langle \texttt{k} \rangle \texttt{tt})$, entonces `P` sí que la cumple.

Nosotros hemos implementado un subconjunto de FULL sin valores. La parte de la lógica con valores de datos requiere un estudio más profundo, y pensamos que para soportarla será necesario conseguir la potencia de algún tipo de demostrador de teoremas. Cierto es que la lógica de reescritura y Maude se han mostrado también muy útiles para desarrollar este tipo de sistemas [Cla00].

La semántica de FULL utiliza los *sucesores* de una expresión de comportamiento después de realizar una acción. Definimos primero operaciones para obtener todas las posibles transiciones de una expresión de comportamiento, que también serán utilizadas por nuestra herramienta para ejecutar o *simular* los procesos. Estas operaciones se definen al metanivel de forma similar a como hicimos con la función `succ` en el Capítulo 5, utilizada para implementar la semántica de Hennessy-Milner, y utilizan el módulo Maude con las reglas de la semántica de LOTOS.

```
fmod SUCC is
  protecting META-LEVEL .

  sort TermSeq .
  subsort Term < TermSeq .
  op mt : -> TermSeq .
  op _+_ : TermSeq TermSeq -> TermSeq [assoc id: mt] .

  var M : Module .
```

```
vars T T' T'' T1 T2 T3 : Term .
var TS : TermSeq .
var N : MachineInt .
```

La operación `transitions` recibe un módulo con la implementación de la semántica (extendido como veremos con la sintaxis y la semántica de las expresiones de datos) y un término t que represente una expresión de comportamiento, y devuelve la secuencia de términos que representan las posibles transiciones de t. Utilizamos la operación `metaSearch` que representa al metanivel el comando `search` utilizado en la Sección 7.3.4.

```
op transitions : Module Term -> TermSeq .
op transitions : Module Term MachineInt -> TermSeq .

eq transitions(M, T) = transitions(M,T,0) .

eq transitions(M, T, N) =
   if metaSearch(M, T, 'X:TCondEventBehExp, nil, '+, 1, N) == failure
   then mt
   else getTerm(metaSearch(M, T, 'X:TCondEventBehExp, nil, '+, 1, N))
         + transitions(M, T, N + 1) fi .
```

La siguiente versión de esta operación, `transitions-subst`, calcula también las transiciones posibles de una expresión de comportamiento, pero aplicando la sustitución de cada transición a la expresión de comportamiento resultado correspondiente.

```
op transitions-subst : Module Term -> TermSeq .
op apply-subst : Module TermSeq -> TermSeq .

eq transitions-subst(M, T) = apply-subst(M, transitions(M, T, 0)) .

eq apply-subst(M, mt) = mt .
eq apply-subst(M, ('`{_`}`{_`}_[T1,T2,T3]) + TS) =
   getTerm(metaReduce(M, '`{_`}`{_`}_[T1,T2,'apply-subst[T1,T3]])) +
   apply-subst(M, TS) .
```

La operación `ttranstions` es la versión correspondiente para transiciones de términos (LOTOS).

```
op ttransitions : Module Term -> TermSeq .
op ttransitions : Module Term MachineInt -> TermSeq .

eq ttransitions(M, T) = ttransitions(M,T,0) .

eq ttransitions(M, T, N) =
   if metaSearch(M, T, 'X:TCondEventTerm, nil, '+, 1, N) == failure
   then mt
   else getTerm(metaSearch(M, T, 'X:TCondEventTerm, nil, '+, 1, N))
         + ttransitions(M, T, N + 1) fi .
```

Ahora podemos definir una operación `succ` que devuelva todos los sucesores de una expresión de comportamiento después de realizar una acción dada. Esta es la operación que será utilizada en la definición de la semántica de la lógica modal. Por ahora nos podemos limitar a utilizar el módulo (metarrepresentado) `LOTOS-SEMANTICS` sin extensiones, dado que en los operadores de la lógica modal no se permiten valores. El término `['LOTOS-SEMANTICS]` denota el módulo `LOTOS-SEMANTICS` metarrepresentado. La operación `succ` con un argumento `T` devuelve todos los sucesores de la expresión de comportamiento que representa `T`, como términos (metarrepresentados) de tipo `TCondEventBehExp`. La operación `succ` con tres argumentos `T`, `T1` y `T2`, devuelve solo los sucesores de `T` tras hacer una transición de LOTOS cuya condición y evento coincidan con `T1` y `T2`, respectivamente.

```
  op succ : Term -> TermSeq .
  op succ : Term Term Term -> TermSeq .
  op filter : TermSeq Term Term -> TermSeq .
  op equal : Term Term -> Bool .


  eq succ(T) = transitions(['LOTOS-SEMANTICS], T, 0) .
  eq succ(T,T1,T2) = filter(succ(T),T1,T2) .

  eq filter(mt, T1, T2) = mt .
  eq filter((''{_'}'{_'}_[T,T',T'']) + TS, T1, T2) =
     if equal(T,T1) and equal(T',T2) then
       T'' + filter(TS,T1,T2)
     else
       filter(TS,T1,T2)
     fi .

  eq equal(T, T') = getTerm(metaReduce(['LOTOS-SEMANTICS],
                                       '_==_[T,T'])) == 'true.Bool .
endfm
```

El siguiente módulo contiene la representación del subconjunto de la lógica FULL que nosotros implementamos. Presenta la sintaxis de FULL y su semántica. El término `P |= Phi` es un booleano, que es `true` si la expresión de comportamiento `P` satisface la fórmula `Phi`. Obsérvese el uso de las operaciones `forall` y `exists` para representar los cuantificadores universal y existencial. El módulo `MOVE-UP` utilizado define una función `up` para metarrepresentar elementos de la sintaxis de LOTOS, equivalente a la función con el mismo nombre de Full Maude.

```
mod FULL is
  protecting LOTOS-SYNTAX .
  protecting SUCC .
  protecting MOVE-UP .
  sort Formula .

  ops tt ff : -> Formula .
```

```
  ops _/\_  _\/_ : Formula Formula -> Formula .
  op <_>_ : SimpleAction Formula -> Formula .
  op [_]_ : SimpleAction Formula -> Formula .

  op forall : TermSeq Formula -> Bool .
  op exists : TermSeq Formula -> Bool .

  op _|=_ : BehaviourExp Formula -> Bool .
  op _|=_ : Term Formula -> Bool .

  var P : BehaviourExp .
  var A : SimpleAction .
  var T : Term .
  var TS : TermSeq .
  vars Phi Psi : Formula .

  eq P |= Phi = up(P) |= Phi .

  eq T |= tt = true .
  eq T |= ff = false .

  eq T |= Phi /\ Psi = (T |= Phi) and (T |= Psi) .

  eq T |= Phi \/ Psi = T |= Phi or  T |= Psi .

  eq T |= [ A ] Phi = forall(succ(T, up(true), up(A)), Phi) .

  eq T |= < A > Phi = exists(succ(T, up(true), up(A)), Phi) .

  eq forall(mt, Phi) = true .
  eq forall(T + TS, Phi) = (T |= Phi) and forall(TS, Phi) .

  eq exists(mt, Phi) = false .
  eq exists(T + TS, Phi) = T |= Phi or exists(TS,Phi) .

endm
```

Veremos ejemplos de utilización de esta lógica en la Sección 7.6.6, cuando tengamos nuestra herramienta completamente construida.

7.5. Traducción de especificaciones ACT ONE

Nuestro objetivo ahora consiste en ser capaces de introducir en nuestra herramienta especificaciones ACT ONE, que serán traducidas internamente a módulos funcionales de Maude.

Comenzaremos introduciendo la sintaxis de ACT ONE. Como ya vimos en la Sección 3.5, en Maude la *definición de sintaxis* de un lenguaje $\mathcal{L}$ se lleva a cabo definiendo un tipo de datos $\texttt{Module}_{\mathcal{L}}$, lo cual puede hacerse de una forma muy flexible, que refleje

fielmente la sintaxis concreta del lenguaje $\mathcal{L}$. Las particularidades al nivel léxico de $\mathcal{L}$ pueden tenerse en cuenta por medio de tipos de datos *burbuja* (cualquier cadena no vacía de identificadores Maude) definidos por el usuario, que correspondan de forma adecuada a las nociones de *token* e *identificador* del lenguaje en cuestión. Las burbujas corresponden a partes de un módulo de un lenguaje que solo pueden ser analizadas sintácticamente una vez que está disponible la gramática introducida por la signatura definida en el propio módulo [CDE+98b]. Esto es especialmente importante cuando el lenguaje $\mathcal{L}$ tiene sintaxis definida por el usuario, como ocurre en nuestro caso con ACT ONE (igual que ocurre con las especificaciones en Maude).

La idea es que la sintaxis de un lenguaje que permite módulos incluyendo características sintácticas definidas por el usuario se puede ver de manera natural como una sintaxis con dos niveles diferentes: la que podemos llamar sintaxis *al nivel más alto* del lenguaje, y la sintaxis definible por el usuario introducida en cada módulo. Los tipos de datos burbujas nos permiten reflejar esta duplicidad de niveles.

Para ilustrar este concepto, veamos la siguiente especificación ACT ONE, adaptación del módulo funcional Maude `NAT3` que ya vimos en la Sección 3.5:

```
type NAT3 is
  sorts Nat3
  opns
    0 : -> Nat3
    s_ : Nat3 -> Nat3
  eqns
    ofsort Nat
      s s s 0 = 0 ;
endtype
```

Las cadenas de caracteres recuadradas no son parte de la sintaxis al nivel más alto de ACT ONE. De hecho, solo pueden analizarse sintácticamente con la gramática asociada a la signatura del módulo `NAT3`.

La gramática de ACT ONE se define en un módulo funcional `ACTONE-SIGN`. A continuación mostramos parte de esta definición.

```
fmod ACTONE-SIGN is

  sort Token NeTokenList Bubble .

  sorts OperDeclList DeclList TypeDecl
        SortName SortNameList EqDecl EqDeclGroup
        VariDeclList VariDeclGroup VarEqDeclList .

  op type_is_endtype : Token DeclList -> TypeDecl  [prec 20 gather(e e)] .

  op sorts_ : Token -> Decl [prec 18 gather(e)] .
  op opns_ : OperDeclList -> Decl [prec 18 gather(e)] .
  op eqns_ : VarEqDeclList -> Decl [prec 18 gather(e)] .
```

```
  op forall_ : VariDeclList -> VariDeclGroup [prec 15 gather(e)] .
  op ofsort__ : Token EqDeclList -> EqDeclGroup [prec 15] .

  op _: ->_ : Token SortName -> OperDecl [prec 5] .
  op _:_->_ : Token SortNameList SortName -> OperDecl [prec 5] .

  op _=_; : Bubble Bubble -> EqDecl [prec 5] .
  op _=>_=_; : Bubble Bubble Bubble -> EqDecl [prec 5] .

endfm
```

En el módulo `META-LOTOS-TOOL-SIGN` al final de la Sección 7.6.1 se puede ver la definición de los tipos `Bubble`. Aunque hemos estado hablando de burbujas de forma genérica, hay de hecho diferentes clases de burbujas. En particular, es ese módulo las burbujas de longitud uno se denominan `Token`. En [CDE+99, CDE+01] se puede encontrar una descripción más detallada de los tipos burbujas y de cómo realizan su función en Maude.

Después de haber definido el módulo con la sintaxis de ACT ONE, podemos utilizar la función `metaParse` del módulo `META-LEVEL`, la cual recibe como argumentos la representación de un módulo M y la representación de una lista de tokens (como valor del tipo `QidList`, como se vio en la Sección 2.2.4) y devuelve la metarrepresentación del término analizado (un árbol sintáctico que puede contener burbujas) según la signatura de M correspondiente a la lista de tokens.

El siguiente paso consiste en definir una operación `translate` que recibe como argumento el término analizado y devuelve un módulo funcional con la misma semántica que la especificación en ACT ONE introducida. El análisis sintáctico de las posibles burbujas se hará también en este segundo paso.

$$\texttt{QidList} \xrightarrow{\texttt{metaParse}} \texttt{Grammar}_{\text{ACT ONE}} \xrightarrow{\texttt{translate}} \texttt{FModule}$$

Con nuestra traducción conseguimos el siguiente resultado.

Conservación de la equivalencia de ACT ONE: Dada una especificación ACT ONE SP, y términos t y t' en SP, tenemos que

$$SP \models t \equiv t' \iff M \models t_M \equiv t'_M$$

donde $M = \texttt{translate}(\texttt{metaParse}(\texttt{ACTONE-GRAMMAR}, SP))$, y t_M y t'_M son las representaciones de t y t' en M.

Antes de ver con más detalle cómo se realiza la traducción a un módulo funcional de Maude, veamos cómo se realizan los dos pasos para el módulo ejemplo anterior `NAT3`. Si ejecutamos la operación `metaParse` con el módulo metarrepresentado `ACTONE-GRAMMAR` (con la sintaxis al nivel más alto de ACT ONE) y la siguiente lista de identificadores con comilla

```
'type 'NAT3 'is
  'sorts 'Nat3
  'opns
    '0 ': '-> 'Nat3
    's_ ': 'Nat3 '-> 'Nat3
  'eqns
    'ofsort 'Nat
      's 's 's '0 '= '0 ';
'endtype
```

obtenemos el siguiente término metarrepresentado, que incluye tanto tokens como burbujas metapresentadas:

```
'type_is_endtype[
  'token[''NAT3.Qid],
  '__['sorts_['token[''Nat3.Qid]],
  '__['opns_[
      '__['_:'->_['token[''0.Qid],'token[''Nat3.Qid]],
          '_:_->_['token[''s_.Qid],'token[''Nat3.Qid],'token[''Nat3.Qid]]]],
      'eqns_[
         'ofsort__['token[''Nat.Qid],
         '_=_;['bubble['__[''s.Qid,''s.Qid,''s.Qid,''0.Qid]],'bubble[''0.Qid]]
                  ]
           ]
     ]
     ]
               ]
```

Los tokens y las burbujas tienen como argumentos listas (unitarias en el primer caso) metarrepresentadas de identificadores con comilla, es decir, valores de tipo `QidList` metarrepresentados. Son estos valores los que hay que volver a analizar sintácticamente (bajándolos un nivel de representación), con la sintaxis definida por el usuario, en el apartado `opns` de la especificación en ACT ONE.

Si ejecutamos la operación `translateType` comentada más abajo sobre el término metarrepresentado anterior, obtenemos el siguiente módulo funcional Maude metarrepresentado, de tipo `FModule`:

```
fmod 'NAT3 is
  including 'DATAEXP-SYNTAX .
  sorts 'Nat3 .
  subsort 'VarId < 'Nat3 .
  subsort 'Nat3 < 'DataExp .
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  eq 's_['s_['s_['0.Nat3]]] = '0.Nat3 .
endfm
```

Este módulo es la traducción de la especificación `NAT3` en ACT ONE vista al comienzo de esta sección.

Veamos ahora cómo se realiza la traducción desde el término devuelto por `metaParse` hasta un módulo funcional en Maude. Primero definimos una operación que extrae la signatura definida por el usuario en la especificación.

```
op extractSignature : Term -> FModule .
```

Esta operación, dentro del módulo `ACTONE-TRANSLATION`, recibe como argumento el término devuelto por `metaParse` y devuelve un módulo funcional que contiene las declaraciones Maude correspondientes a los tipos y operaciones encontradas en dicho término. Este módulo se utilizará para analizar las burbujas.

Las siguientes funciones realizan la traducción propiamente dicha:

```
op translateType : Term -> FModule .
op translateType : Term FModule FModule -> FModule .
op translateDeclList : Term FModule FModule -> FModule .
```

La primera operación es la principal. Recibe como argumento el término devuelto por `metaParse` y devuelve su traducción como módulo funcional. Para ello utiliza la operación `translateType` con tres argumentos: la parte de especificación ACT ONE aún no traducida, el módulo Maude con la traducción ya hecha, y el módulo Maude con (solo) la signatura de la parte ya traducida.

```
vars Q QI V S : Qid .
vars T T' T'' T''' : Term .
vars M M' : Module .
var VS : VarSet .
var QIL : QidList .

eq translateType(T) =
   translateType(T,
                 addImportList(including 'DATAEXP-SYNTAX ., emptyFModule),
                 emptyFModule) .

eq translateType('__[T,T'], M, M') =
    translateType(T', translateType(T, M, M'),
                      addDecls(M', extractSignature(T))) .

eq translateType('type_is_endtype['token[T'],T''], M, M') =
   translateDeclList(T'', M,
     addDecls(M',extractSignature('type_is_endtype['token[T'],T'']))) .

eq translateDeclList('__[T,T'], M, M') =
   translateDeclList(T', translateDeclList(T,M, M'), M') .
```

La operación `translateDeclList` recorre la lista de declaraciones dentro de la declaración de un tipo, y va añadiendo a su segundo argumento la traducción de cada declaración. Veamos, a continuación, cómo se traducen algunas de estas declaraciones.

Cuando en la especificación en ACT ONE aparece una declaración de un tipo `T`, esta no solo produce su traducción a una declaración de tipo en Maude para el tipo `T`, sino que también declara el tipo `T` como subtipo del tipo `DataExp` (ya que los valores del nuevo tipo podrían ser utilizados en una comunicación dentro de una expresión de comportamiento) e introduce además el tipo de las variables de LOTOS `VarId` como subtipo del tipo `T` (ya que se podrían utilizar variables de LOTOS para construir valores del tipo `T`). Obtenemos así una herramienta general en Maude por medio de la cual se pueden introducir y ejecutar especificaciones en ACT ONE, que nosotros utilizaremos para integrar los módulos de ACT ONE con especificaciones en LOTOS.

```
eq translateDeclList('sorts_['token[T]], M, M') =
   addSubsortDeclSet(subsort downQid(T) < 'DataExp .,
     addSubsortDeclSet(subsort 'VarId < downQid(T) .,
       addSortSet(downQid(T), M))) .
```

La operación

```
op translateEqDeclList : Term FModule FModule VarSet -> FModule .
```

traduce una lista de declaraciones de ecuaciones. El cuarto argumento es un conjunto de variables de ACT ONE que pueden aparecer en las ecuaciones (es decir, la ecuación se encuentra en el *ámbito de visibilidad* de dichas variables). Para poder analizar de forma correcta las burbujas en la ecuación, la información relacionada con las variables tiene que ser incluida en la expresión como variables de Maude. Recuérdese que en Maude 2.0 una variable metarrepresentada es un identificador con comilla que incluye el nombre de la variable y su tipo, separados por ':'. Las siguientes ecuaciones describen cómo se realiza la traducción de una ecuación (posiblemente condicional).

```
eq translateEqDeclList('_=_;['bubble[T], 'bubble[T']], M, M', VS) =
   addEquationSet(
       (eq getTerm(metaParse(M', insertVar(downQidList(T), VS), anyType))
         = getTerm(metaParse(M', insertVar(downQidList(T'), VS), anyType)) .),
      M) .

eq translateEqDeclList('_=>_=_;['bubble[T], 'bubble[T'], 'bubble[T'']],
                       M, M', VS) =
   addEquationSet(
       (ceq getTerm(metaParse(M', insertVar(downQidList(T'), VS), anyType))
         = getTerm(metaParse(M', insertVar(downQidList(T''), VS), anyType))
         if getTerm(metaParse(M', insertVar(downQidList(T), VS), anyType))
            = 'true.Bool .),
      M) .
```

A continuación mostramos un ejemplo de cómo una especificación en ACT ONE, algo más completa que la especificación `NAT3` vista anteriormente, se traduce a un módulo funcional en Maude. La especificación en ACT ONE de la izquierda se traduce al módulo funcional en Maude de la derecha:

```
type Naturals is
  sorts Nat
  opns
    0 : -> Nat
    s : Nat -> Nat
    _+_ : Nat, Nat -> Nat
  eqns
    forall x, y : Nat
    ofsort Nat
      0 + x = x ;
      s(x) + y = s(x + y) ;
endtype
```

$\Longrightarrow$

```
fmod Naturals is
  including DATAEXP-SYNTAX .
  sorts Nat .
  subsort VarId < Nat .
  subsort Nat < DataExp .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  eq 0 + x:Nat = x:Nat .
  eq s(x:Nat) + y:Nat =
        s(x:Nat + y:Nat) .
endfm
```

7.5.1. Extensión de los módulos

En la Sección 7.3.2 vimos que la operación que realiza la sustitución sintáctica y la operación que extrae las variables que aparecen en una expresión de comportamiento no fueron definidas completamente. La razón fue la misma en los dos casos: la presencia de expresiones de datos con sintaxis definida por el usuario y, por tanto, desconocida en aquel momento.

Ahora que conocemos la especificación ACT ONE, y que esta se ha traducido a un módulo funcional, podemos definir estas operaciones sobre expresiones de datos que utilicen la nueva sintaxis. Debido a las facilidades de metaprogramación que ofrece Maude, podemos hacerlo de forma automática. En el módulo `MODULE-EXTENSIONS` mostrado a continuación, definimos operaciones que toman un módulo M y devuelven el mismo módulo M extendido con ecuaciones que definen la sustitución y la extracción de variables sobre expresiones construidas utilizando la signatura en el módulo M.

Por ejemplo, si la operación `addOpervars` (descrita más abajo) se aplica al módulo `Naturals` visto anteriormente, se añaden las siguiente ecuaciones:[4]

```
eq vars(0) = mt .
eq vars(s(v1:Nat)) = vars(v1:Nat) .
eq vars(v1:Nat + v2:Nat) = vars(v1:Nat) U vars(v2:Nat) .
```

Obsérvese que esta no es la forma más natural de definir esta operación, ya que los únicos constructores del tipo `Nat` son `0` y `s`, por lo que podrían parecer suficientes las dos primeras ecuaciones. Sin embargo, aquí estamos definiendo la operación sobre expresiones que pueden contener variables LOTOS, por lo que la tercera ecuación es también necesaria.

Veamos cómo especificar la operación `addOpervars`. La operación recibe como argumento un módulo M correspondiente a la traducción de una especificación ACT ONE. Por tanto, las operaciones declaradas en este módulo M pueden utilizarse para construir expresiones LOTOS de cierto tipo. La función `addOpervars` recorre la lista de declaraciones de operadores y para cada uno de ellos añade una ecuación que indica cómo se extraen las variables de un término cuyo operador más externo sea aquel. El módulo `UNIT` utilizado

[4] Recuérdese que en Maude 2.0 las variables no tienen que ser declaradas explícitamente mediante declaraciones de variables, aunque por conveniencia estas se permiten.

incluye operaciones para la construcción de módulos metarrepresentados a partir de sus componentes: declaración de tipos, operaciones, ecuaciones, reglas, etc.

```
fmod MODULE-EXTENSIONS is
  protecting UNIT .

  op addOpervars : Module -> Module .
  op addOpervars : OpDeclSet Module -> Module .
  op addOpervars : Qid TypeList Qid Module -> Module .
  op buildArgs : TypeList MachineInt -> TermList .
  op buildArgs2 : Qid TermList -> TermList .

  var M : Module .
  vars OP S A A' : Qid .
  var ARGS : TypeList .
  var T : Term .
  var TL : TermList .
  var AttS : AttrSet .
  var ODS : OpDeclSet .
  var N : MachineInt .

  eq addOpervars(M) = addOpervars(opDeclSet(M), M) .

  eq addOpervars(none, M) = M .
  eq addOpervars(op OP : ARGS -> S [AttS] . ODS, M) =
      addOpervars(ODS, addOpervars(OP, ARGS, S, M)) .

  eq addOpervars(OP, nil, S, M) =
      addEquationSet(eq 'vars[conc(OP,conc('.,S))] = 'mt.VarSet ., M) .
  eq addOpervars(OP, A ARGS, S, M) =
      addEquationSet(eq 'vars[OP[buildArgs(A ARGS, 1)]] =
         if ARGS == nil then 'vars[buildArgs(A ARGS, 1)]
         else '_U_[buildArgs2('vars, buildArgs(A ARGS, 1))] fi ., M) .

  eq buildArgs(A, N) = conc(conc(index('v,N),':), A) .
  eq buildArgs(A A' ARGS, N) = buildArgs(A, N), buildArgs(A' ARGS, N + 1) .
  eq buildArgs2(OP, T) = OP[T] .
  eq buildArgs2(OP, (T,TL)) = OP[T], buildArgs2(OP,TL) .
endfm
```

Las ecuaciones añadidas por la operación `addOpervars` junto con la ecuación

```
  eq vars(x) = x .
```

que vimos en la Sección 7.3.2 (página 184) indican cómo extraer las variables de una expresión de datos en LOTOS definida con la sintaxis del usuario.

De igual forma se define la operación

```
 op addOperSubs : Module -> Module .
```

que añade a un módulo ecuaciones que definen la operación de sustitución de una expresión por otra sobre términos construidos con las operaciones del módulo.

7.6. Construcción de la herramienta LOTOS

Queremos construir una herramienta formal donde se puedan introducir y ejecutar especificaciones LOTOS completas, que incluyan una especificación de tipos de datos ACT ONE, una expresión de comportamiento principal y definiciones de procesos. Para poder *ejecutar* o *simular* la especificación, queremos ser capaces de recorrer el sistema de transiciones simbólico generado para la expresión de comportamiento principal utilizando la semántica simbólica instanciada con los tipos de datos concretos dados en la especificación ACT ONE y las definiciones de procesos dadas.

7.6.1. Gramática de la interfaz de la herramienta

Comenzamos definiendo la signatura (o sintaxis concreta) de LOTOS y la signatura de los comandos que vamos a utilizar en nuestra herramienta para trabajar con la especificación introducida. Hay una separación importante entre la signatura concreta utilizada por los usuarios para escribir sus propias especificaciones (utilizada por la herramienta para analizar sintácticamente dichas especificaciones y definida utilizando conceptos como los tipos burbujas) y la sintaxis abstracta que definimos en la Sección 7.3.1 en el módulo `LOTOS-SYNTAX` (utilizada para representar de forma abstracta términos de LOTOS), aunque aparentemente sean bastante similares.

La signatura de ACT ONE se presentó en el módulo `ACTONE-SIGN` y la de LOTOS se presenta parcialmente, a continuación, en el módulo `LOTOS-SIGN`:

```
fmod LOTOS-SIGN is
  protecting ACTONE-SIGN .

  sorts VarId SortId GateId ProcId .
  subsorts Token < VarId SortId GateId ProcId .

  sort DataExp .
  subsort Bubble < DataExp .

  sorts BehaviourExp Offer StrucAction IdDecl GateIdList SelecPred .

  op stop : -> BehaviourExp .
  op exit : -> BehaviourExp .
  op __ : GateId Offer -> StrucAction [prec 30 gather(e e)] .

  op !_ : Bubble -> Offer [prec 25] .
  op ?_ : IdDecl -> Offer [prec 25] .
  op _:_ : VarId SortId -> IdDecl [prec 20] .

  op _;_ : Action BehaviourExp -> BehaviourExp [prec 35] .
```

```
  op _‘[‘]_ : BehaviourExp BehaviourExp -> BehaviourExp [assoc prec 40] .
  op hide_in_ : GateIdList BehaviourExp -> BehaviourExp [prec 40] .
  op ‘[_‘]->_ : SelecPred BehaviourExp -> BehaviourExp [prec 40] .

endfm
```

El siguiente módulo, `LOTOS-TOOL-SIGN`, incluye las signaturas de ACT ONE y LOTOS y define los comandos de nuestra herramienta.

```
fmod LOTOS-TOOL-SIGN is
  protecting LOTOS-SIGN .
  protecting MACHINE-INT .

  sort LotosCommand .

  op show process . : -> LotosCommand .
  op show transitions . : -> LotosCommand .
  op show transitions of_. : BehaviourExp -> LotosCommand .
  op cont_. : MachineInt -> LotosCommand .
  op cont . :  -> LotosCommand .
  op show state . : -> LotosCommand .
  op show term transitions . : -> LotosCommand .
  op show term transitions of_. : BehaviourExp -> LotosCommand .
  op tcont_. : MachineInt -> LotosCommand .
  op tcont . :  -> LotosCommand .
  op sat_. : Formula -> LotosCommand .
endfm
```

El primer comando se utiliza para mostrar el proceso actual, es decir, la expresión de comportamiento utilizada por defecto si en el resto de comandos se utiliza la versión que no incluye este argumento. El segundo y tercer comandos se utilizan para mostrar las posibles transiciones (definidas por la semántica simbólica) del proceso actual (por defecto) o del dado explícitamente, de modo que con estos comandos se comienza la ejecución de un proceso. El cuarto comando se utiliza para proseguir con la ejecución siguiendo una de las posibles transiciones siguientes, en concreto la indicada en el argumento del comando. El comando `cont` (sin número) es equivalente a `cont 1`. El sexto comando se utiliza para mostrar el *estado* actual de la ejecución, es decir, la condición actual, la traza ejecutada y las posibles transiciones siguientes. Los cuatro comandos siguientes corresponden a las transiciones de términos LOTOS, en vez de transiciones de expresiones de comportamiento. El último comando sirve para pedir a la herramienta que compruebe si el proceso actual cumple la fórmula dada (del subconjunto implementado) de la lógica FULL.

Para poder analizar sintácticamente una entrada utilizando la función predefinida `metaParse`, necesitamos dar la metarrepresentación de la signatura dentro de la cual se tiene que hacer el análisis sintáctico. Incluyendo el módulo anterior, `LOTOS-TOOL-SIGN`, dentro del módulo metarrepresentado `LOTOS-GRAMMAR` mostrado a continuación, obtenemos la metarrepresentación de `LOTOS-TOOL-SIGN`. El módulo `LOTOS-GRAMMAR` será utilizado en las llamadas a la función `metaParse`, para conseguir que la entrada sea analizada con esta

signatura[5]. Obsérvese que la llamada a `metaParse` nos devolverá un término que representará el árbol sintáctico de la entrada (posiblemente incluyendo burbujas). Este término será transformado entonces a un término de un tipo de datos apropiado.

```
fmod META-LOTOS-TOOL-SIGN is
  including META-LEVEL .

  op LOTOS-GRAMMAR : -> FModule .
  eq LOTOS-GRAMMAR
    = (fmod 'LOTOS-GRAMMAR is
        including 'QID-LIST .
        including 'LOTOS-TOOL-SIGN .
        sorts none .
        none
        op 'token : 'Qid -> 'Token
             [special(
               (id-hook('Bubble, '1 '1)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)))] .
        op 'bubble : 'QidList -> 'Bubble
             [special(
               (id-hook('Bubble, '1 '-1 ''( ''))
                op-hook('qidListSymbol, '__, 'QidList 'QidList, 'QidList)
                op-hook('qidSymbol, '<Qids>, nil, 'Qid)
                id-hook('Exclude, '. '! '=> '; 'any 'ofsort ''[ ''] )))] .
        none
        none
     endfm) .

endfm
```

7.6.2. Procesamiento de la entrada LOTOS

Cuando se introducen expresiones de comportamiento LOTOS, bien como parte de la especificación completa o bien en un comando de la herramienta, tienen que ser transformadas en elementos del tipo `BehaviourExp` del módulo `LOTOS-SYNTAX` (Sección 7.3.1). El árbol sintáctico devuelto por `metaParse` utilizando el módulo `LOTOS-GRAMMAR` puede contener burbujas (en los lugares donde aparecen expresiones de datos) y tokens (en los lugares donde aparecen nuevos identificadores, como los que nombran puertas o variables) que tienen que ser analizados de nuevo utilizando la definición de sintaxis introducida por el usuario. Esta sintaxis se obtiene traduciendo los tipos definidos en ACT ONE en módulos funcionales, como se explicó anteriormente. Es más, una expresión de comportamiento por sí misma puede definir nueva sintaxis, ya que puede declarar nuevas variables LOTOS

[5] Aunque en las secciones anteriores hemos utilizado también una constante `ACTONE-GRAMMAR` para referirnos al módulo con la signatura de ACT ONE, aquí definimos una única constante `LOTOS-GRAMMAR` que incluye tanto la signatura de ACT ONE como la de LOTOS. Esto es debido a que en nuestra herramienta las especificaciones se introducirán de forma completa, incluyendo tanto la parte ACT ONE como la parte LOTOS.

por medio de comunicaciones de valores del tipo `?`, que en lo sucesivo pueden aparecer en las expresiones. Por ejemplo, cuando se procese la expresión de comportamiento

```
g ? x : Nat ; h ! (s(x) + s(0)) ; stop
```

la expresión de datos `s(x) + s(0)` debe ser analizada sintácticamente en un módulo donde estén definidos los operadores `0`, `s` y `_+_`, y utilizando el hecho de que `x` es una variable de tipo `Nat`.

A continuación vamos a ver algunas de las operaciones y ecuaciones que definen esta traducción.

Utilizamos la operación `parseProcess` para realizar la traducción descrita arriba. Esta operación recibe como argumentos el término devuelto por `metaParse` que representa una expresión de comportamiento, el módulo metarrepresentado con la sintaxis de los tipos de datos (obtenida a partir de la especificación ACT ONE) y el conjunto de variables libres que pueden aparecer en la expresión de comportamiento. La operación devuelve una expresión de comportamiento (metarrepresentada) sin burbujas. Para ello utiliza la operación `parseAction` que devuelve el término que metarrepresenta la acción dada, sin burbujas, y las variables declaradas en la acción, si hubiera alguna.

```
op parseAction : Term Module VarSet -> TermVars .
op parseProcess : Term Module VarSet -> Term .
op parseDataExp : Term Module VarSet -> Term .
op parseOffer : Term Module VarSet -> TermVars .
op parseProcDeclList : Term Module -> Term .
```

Obsérvese cómo la siguiente ecuación transforma un *token* que representa una puerta, en un identificador de puerta, utilizando la sintaxis abstracta del módulo `LOTOS-SYNTAX` de la Sección 7.3.1:

```
eq parseAction('token[G], M, VS) = < 'G[G], mt > .
eq parseAction('i.SimpleAction, M, VS) = < 'i.SimpleAction, mt > .
eq parseAction('__[G, T'], M, VS) =
    < '__[term(parseAction(G, M, VS)), term(parseOffer(T',M, VS))],
      vars(parseOffer(T',M, VS)) > .
```

Cuando se analiza sintácticamente un proceso prefijado por una acción, primero se analiza la acción y después se analiza el proceso, utilizando un conjunto de variables extendido con las variables nuevas declaradas por la acción (si hubiera alguna):

```
eq parseProcess('_;_[T,T'],M,VS) =
    '_;_[term(parseAction(T,M,VS)),
         parseProcess(T',M, VS vars(parseAction(T,M,VS)))] .
```

Hay otros operadores, como el operador de elección sobre valores, que también declaran nuevas variables. Su tratamiento es el mismo: la expresión de comportamiento interna (`T''` en la siguiente ecuación) se analiza con un conjunto de variables extendido con la nueva variable:

```
eq parseProcess('choice_'[']_['_:_['token[T],'token[T']],T''], M, VS) =
    'choice_'[']_['_:_['V[T], 'S[T']],
                  parseProcess(T'',M, VS (downQid(T) : downQid(T')))] .
```

La operación `parseDataExp` recibe una burbuja que representa una expresión, un módulo con la sintaxis con la cual se tiene que analizar la expresión, y un conjunto de variables de LOTOS que pueden aparecer en la expresión. Para poder analizar de forma correcta esta burbuja, las variables LOTOS que aparezcan en la expresión original tienen que sustituirse por variables de Maude asociadas. El término resultante contendrá dichas variables de Maude, que tendrán que ser reconvertidas en las variables de LOTOS, pero utilizando esta vez la sintaxis abstracta, con lo que obtenemos términos de la forma `V(Q)`, siendo `Q` un identificador con comilla.

```
eq parseDataExp('bubble[E], M, VS) =
   varsMaudeToLotos(getTerm(metaParse(M,
       insertVar(downQidList(E),VS),anyType))) .
```

La operación `parseProcDeclList` construye un contexto LOTOS (metarrepresentado) que incluye las definiciones de los procesos declarados en una especificación.

```
eq parseProcDeclList('emptyProcList.ProcDeclList,M) = 'nil.Context .
eq parseProcDeclList('process_'[_']'(_'):_:=_endproc['token[T1],
                                                      T2,T3,T4,T5],M) =
   'process['P[T1], parseGateIdList(T2), parseVariDeclList(T3),
            parseProcess(T5, M, varDeclToVarSet(T3)) ] .
eq parseProcDeclList('__[T,T'], M) =
   '_&_[parseProcDeclList(T, M), parseProcDeclList(T', M)] .
```

Las fórmulas de la lógica modal FULL que se escriben en el comando `sat` de nuestra herramienta también tienen que ser traducidas, desde la sintaxis concreta a la sintaxis abstracta en la que los identificadores de puertas son de la forma `G(Q)`, siendo `Q` un identificador con comilla.

```
op parseFormula : Term -> Term .

eq parseFormula(C) = C .
eq parseFormula('_/\_[T,T']) = '_/\_[parseFormula(T),parseFormula(T')] .
eq parseFormula('_\/_[T,T']) = '_\/_[parseFormula(T),parseFormula(T')] .
eq parseFormula('`[_']_['token[G],T']) = '`[_']_['G[G], parseFormula(T')] .
eq parseFormula('<_>_['token[G],T']) = '<_>_['G[G], parseFormula(T')] .
```

7.6.3. Procesamiento de los comandos de la herramienta

El siguiente módulo define operaciones auxiliares utilizadas para el procesamiento de los comandos de la herramienta. Estas operaciones serán utilizadas por las reglas que definen el tratamiento del estado interno de la herramienta (véase Sección 7.6.4). Sirven para mostrar por pantalla de forma correcta las transiciones de un proceso, y para seleccionar los distintos argumentos de un término (metarrepresentado) de tipo `TCondEventBehExp`.

```
fmod LOTOS-COMMAND-PROCESSING is
  protecting SUCC .

  vars Q Q1 Q2 : Qid .
  var QIL : QidList .
  var M : Module .
  vars T T1 T2 T3 : Term .
  var TS : TermSeq .
  var N : MachineInt .

  op meta-pretty-print-transitions : Module TermSeq -> QidList .
  op meta-pretty-print-transitions : Module TermSeq MachineInt -> QidList .
  op meta-pretty-print-trace : Module Term -> QidList .
  op meta-pretty-print-condition : Module Term -> QidList .

  eq meta-pretty-print-transitions(M, TS) =
    if TS == mt then ('\n 'No 'more 'transitions '. '\n)
    else
      ('\n 'TRANSITIONS ': '\n
       meta-pretty-print-transitions(M, TS, 1) '\n )
    fi .

  eq meta-pretty-print-transitions(M, mt, N) = nil .
  eq meta-pretty-print-transitions(M, T + TS, N) =
    '\n conc(index(' , N), '.) '\t filter(metaPrettyPrint(M,T))
       meta-pretty-print-transitions(M, TS, N + 1) .

  eq meta-pretty-print-trace(M, T) =
       ('\n 'Trace ': filter(metaPrettyPrint(M,T)) '\n) .

  eq meta-pretty-print-condition(M, T) =
       ('\n 'Condition ': filter(metaPrettyPrint(M,T)) '\n) .
```

La operación `selectSol` recibe como argumentos un entero `N` y una secuencia de soluciones devueltas por la operación `transitions`, y devuelve el elemento `N`-ésimo de la secuencia.

```
  op selectSol : MachineInt TermSeq -> Term .
  op selectProc : Term -> Term .
  op selectEvent : Term -> Term .
  op selectCond : Term -> Term .

  eq selectSol(1, T + TS) = T .
  ceq selectSol(N, T + TS) = selectSol(_-_(N,1), TS) if N > 1 .

  eq selectProc('`{_`}`{_`}_[T1,T2,T3]) = T3 .
  eq selectEvent('`{_`}`{_`}_[T1,T2,T3]) = T2 .
  eq selectCond('`{_`}`{_`}_[T1,T2,T3]) = T1 .

endfm
```

7.6.4. Tratamiento del estado de la herramienta

En nuestra herramienta, el estado persistente del sistema viene dado por un único objeto que mantiene el estado de la herramienta. Este objeto tiene los siguientes atributos:

- `semantics`, para mantener el módulo actual en el que se pueden ejecutar las expresiones de comportamiento. Se trata pues del módulo `LOTOS-SEMANTICS` de la Sección 7.3.2 extendido con la sintaxis y semántica de las nuevas expresiones de datos;
- `lotosProcess`, para mantener la expresión de comportamiento que etiqueta el nodo del sistema de transiciones simbólico que ha sido alcanzado durante la ejecución;
- `transitions`, para mantener el conjunto de posibles transiciones a partir del proceso en el atributo `lotosProcess`;
- `trace`, para mantener la secuencia de eventos realizados en el camino desde la raíz del STS hasta el nodo actual;
- `condition`, para mantener la conjunción de las condiciones de las transiciones en ese camino; e
- `input` y `output`, para tratar la comunicación con el usuario.

Declaramos la clase siguiente utilizando la notación para clases en los módulos orientados a objetos:

```
class ToolState | semantics : Module,
                  lotosProcess : Term,
                  transitions : TermSeq,
                  trace : Term,
                  condition : Term,
                  input : TermList,
                  output : QidList .
```

Sin embargo, ya que estamos utilizando Core Maude (y no Full Maude), las declaraciones orientadas a objetos no pueden ser tratadas directamente. Debemos usar en cambio las declaraciones equivalentes que eliminan el azúcar sintáctico de las declaraciones orientadas a objetos, como se explica en [Dur99].

```
mod TOOL-STATE-HANDLING is
  protecting CONFIGURATION .
  protecting LOTOS-PARSING .
  protecting LOTOS-COMMAND-PROCESSING .
  protecting ACTONE-TRANSLATION .
  protecting MODULE-EXTENSIONS .

  sort ToolStateClass .
```

```
subsort ToolStateClass < Cid .
op ToolState : -> ToolStateClass .

*** Attributes of the tool state
op semantics :_ : Module -> Attribute .
op lotosProcess :_ : Term -> Attribute .
op transitions :_ : TermSeq -> Attribute .
op trace :_ : Term -> Attribute .
op condition :_ : Term -> Attribute .
op input :_ : TermList -> Attribute .
op output :_ : QidList -> Attribute .

op SYN : -> FModule .
op SEM : -> Module .
eq SYN = ['DATAEXP-SYNTAX] .
eq SEM = ['LOTOS-SEMANTICS] .

var TL : TermList .
var Atts : AttributeSet .
var X@ToolState : ToolStateClass .
var O : Oid .
vars T T' T'' T''' T1 T2 T3 T4 : Term .
vars SynM SemM : Module .
var TS : TermSeq .

op nilTermList : -> TermList .
eq (nilTermList, TL) = TL .
eq (TL, nilTermList) = TL .
```

Las siguientes reglas describen el comportamiento de la herramienta cuando se introduce en el sistema una especificación LOTOS o alguno de los diferentes comandos.

La primera regla procesa una especificación LOTOS introducida en el sistema. Se permiten especificaciones LOTOS con cuatro argumentos: el nombre de la especificación, una especificación ACT ONE que defina los tipos de datos a utilizar, la expresión de comportamiento principal, y una lista de definiciones de procesos (tanto la especificación ACT ONE como la lista de procesos pueden ser vacías). No se permiten declaraciones locales. Cuando se introduce una especificación LOTOS, el atributo `semantics` se actualiza con un nuevo módulo construido de la siguiente manera: primero, la parte ACT ONE de la especificación se traduce a un módulo funcional; después, se le añaden ecuaciones que definen la sustitución sintáctica y la extracción de variables (como se explicó en la Sección 7.5.1); el módulo resultante se une con la metarrepresentación del módulo `LOTOS-SEMANTICS` (con las reglas de la semántica simbólica); y, finalmente, se añade una ecuación que define la constante `context` con las definiciones de procesos dadas en la especificación. El atributo `lotosProcess` también es actualizado con la expresión de comportamiento principal introducida en la especificación, y el resto de los atributos son inicializados. Obsérvese el mensaje colocado en el canal de salida `output`.

```
rl [spec] :
    < O : X@ToolState |
      input : ('specification__behaviour_where_endspec['token[T],
                                                        T',T'',T''']),
      output : nil,
      semantics : SemM, lotosProcess : T1,
      transitions : TS,
      trace : T3,
      condition : T4, Atts >
   => < O : X@ToolState | input : nilTermList,
         output : ('\n 'Introduced 'specification getName(T) '\n),
         semantics : addEquationSet(eq 'context.Context =
                          parseProcDeclList(T''',
                            addDecls(translateType(T'), SYN)) .,
                       addDecls(SEM, addOperSubs(
                                       addOpervars(translateType(T'))))),
         lotosProcess : parseProcess(T'',
                           addDecls(translateType(T'), SYN),mt),
         transitions : mt,
         trace : 'nil.Trace,
         condition : 'true.Bool, Atts > .
```

Obsérvese también cómo se utiliza la variable `Atts` de tipo `AttributeSet` para capturar "el resto" de atributos. Aunque en el objeto de la parte izquierda de la regla aparecen explícitamente todos sus atributos, la regla también podría utilizarse si añadiéramos más atributos al objeto siempre que estos no tuvieran que ser modificados a la hora de introducir una especificación.

Los comandos se tratan mediante reglas similares. Por ejemplo, la siguiente regla trata el comando `show transitions`;

```
rl [show-transitions] :
   < O : X@ToolState |
      input : 'show`transitions`..LotosCommand,
      semantics : SemM,
      lotosProcess : T,
      transitions : TS, Atts >
   => < O : X@ToolState |
         input : 'show`state`..LotosCommand,
         semantics : SemM,
         lotosProcess : T,
         transitions : transitions-subst(SemM,T), Atts >  .
```

Para modificar el valor del atributo `transitions` se utiliza `transitions-subst`, operación definida en la Sección 7.4. Obsérvese cómo en el atributo de entrada se coloca el comando `show state` para que se ejecute la regla que trata este comando, mostrándose por pantalla el estado actual.

```
rl [show-state] :
   < O : X@ToolState |
      input : 'show`state`..LotosCommand,
      output : nil,
      semantics : SemM,
      lotosProcess : T',
      transitions : TS, trace : T1, condition : T2, Atts >
   => < O : X@ToolState |
      input : nilTermList,
      output :
         (meta-pretty-print-trace(SemM, T1)
          meta-pretty-print-condition(SemM, T2)
          meta-pretty-print-transitions(SemM, TS)),
      semantics : SemM,
      lotosProcess : T',
      transitions : TS, trace : T1, condition : T2, Atts > .
```

Las siguientes dos reglas tratan el comando `cont`. La primera se limita a modificar la entrada para que, a continuación, se ejecute la segunda. En esta se modifican los atributos `lotosProcess`, `trace` y `condition` de forma adecuada, dependiendo del número de transición con la que se quiera continuar, y se pide que se muestren las transiciones del nuevo proceso actual:

```
rl [continue] :
   < O : X@ToolState |
      input : 'cont`..LotosCommand, Atts >
   => < O : X@ToolState |
         input : ('cont_.['1.NzMachineInt]), Atts > .

rl [continue] :
   < O : X@ToolState |
      input : ('cont_.[T]), output : nil,
      semantics : SemM,
      lotosProcess : T',
      transitions : TS, trace : T'', condition : T''', Atts >
   => < O : X@ToolState |
      input : 'show`transitions`..LotosCommand, output : nil,
      semantics : SemM,
      lotosProcess : selectProc(selectSol(downMachineInt(T),TS)),
      transitions : TS,
      trace : getTerm(metaReduce(SemM,
               '__[T'',selectEvent(selectSol(downMachineInt(T),TS))])),
      condition : getTerm(metaReduce(SemM,
                   '_/\_[T''',selectCond(
                               selectSol(downMachineInt(T),TS))])), Atts > .
```

Por último, la siguiente regla trata el comando `sat` que pide comprobar si el proceso actual cumple una fórmula dada de la lógica modal FULL:

```
  rl [sat] :
     < O : X@ToolState |
        input : ('sat_.[T]), output : nil,
        lotosProcess : T', Atts >
     => < O : X@ToolState |
        input : nilTermList,
        output : if getTerm(metaReduce(['FULL], '_|=_[T',parseFormula(T)]))
                    == 'true.Bool then
                 'Satisfied '.
                 else 'Not 'Satisfied '.
                 fi ,
        lotosProcess : T', Atts > .

endm
```

7.6.5. El entorno de la herramienta LOTOS

La entrada/salida de especificaciones y de los comandos se lleva a cabo a través del módulo predefinido `LOOP-MODE` [CDE+99], que proporciona un bucle genérico de lectura-evaluación-escritura. Como se explicó en la Sección 2.2.6, este módulo define un operador `[_,_,_]` que puede verse como un objeto persistente con un canal de entrada y otro de salida (su primer y tercer argumento, respectivamente), y un estado (dado por su segundo argumento). Tenemos una flexibilidad total a la hora de definir este estado. En nuestra herramienta, utilizaremos al efecto un objeto de la clase `ToolState`. Cuando se introduce en la entrada de Maude un cierto dato escrito entre paréntesis, el sistema lo coloca en el primer argumento del objeto persistente, como una lista de identificadores con comilla. A continuación se analizará sintácticamente la entrada utilizando la gramática adecuada, y el término resultante se colocará en el atributo `input` del objeto correspondiente al estado de la herramienta. Por último, las reglas vistas en la sección anterior lo procesarán. La salida se trata en el sentido inverso, de modo que la lista de identificadores con comilla, colocada en el tercer argumento del objeto persistente, se imprime en el terminal.

En el siguiente módulo se definen las reglas para inicializar el bucle, y para especificar la comunicación entre este bucle (la entrada/salida del sistema) y el estado persistente del sistema:

```
mod LOTOS is
  including LOOP-MODE .
  protecting TOOL-STATE-HANDLING .
  protecting META-LOTOS-TOOL-SIGN .
```

El estado del sistema viene representado por un único objeto.

```
  subsort Object < State .

  op o : -> Oid .          *** constant object identifier
  op init : -> System .    *** initial state of the persistent object
```

```
var Atts : AttributeSet .
var X@ToolState : ToolStateClass .
var O : Oid .
var Q : Qid .
vars QIL QIL' QIL'' : QidList .
```

La regla `init` inicializa el objeto persistente como un objeto de la clase `ToolState` inicializando todos sus atributos.

```
rl [init] : init
         => [nil,
             < o : ToolState |
                input : nilTermList,
                output : nil,
                semantics : ['LOTOS-SEMANTICS],
                lotosProcess : 'stop.BehaviourExp,
                transitions : mt,
                trace : 'nil.Trace,
                condition : 'true.TransCond >,
             ('\n '\t  'LOTOS 'in 'Maude 'version '1.0 '\n )] .
```

La regla `in` utiliza el módulo `LOTOS-GRAMMAR` para analizar sintácticamente las especificaciones de Full LOTOS y los comandos de la herramienta. Si la entrada es sintácticamente válida, su análisis se coloca en el atributo `input`; en caso contrario, se coloca un mensaje de error en el canal de salida. Obsérvese que la regla solo se ejecuta cuando la (lista de) entrada es no vacía.

```
crl [in] :
      [QIL,
        < O : X@ToolState | input : nilTermList,
                            output : nil, Atts >,
       QIL']
   => if not(metaParse(LOTOS-GRAMMAR, QIL, anyType) :: ResultPair)
      then [nil,
            < O : X@ToolState |
               input : nilTermList, output : nil, Atts >,
            QIL' ('ERROR: 'incorrect 'input '.)]
      else [nil,
            < O : X@ToolState |
               input : getTerm(metaParse(LOTOS-GRAMMAR, QIL, anyType)),
               output : nil, Atts >,
            QIL']
      fi
      if QIL =/= nil .
```

Cuando el atributo `output` del objeto estado de la herramienta contiene una lista no vacía de identificadores con comilla, la siguiente regla la desplaza al tercer argumento del bucle de entrada/salida. Entonces el sistema Maude la muestra por el terminal:

```
  crl [out] :
        [QIL,
         < O : X@ToolState | output : QIL', Atts >,
         QIL'']
     => [QIL,
         < O : X@ToolState | output : nil, Atts >,
         (QIL' QIL'')]
        if QIL' =/= nil .
endm
```

Al sistema se le indica qué objeto tiene que utilizar mediante el siguiente comando:

```
Maude> loop init .
```

Después de introducir este último módulo ya puede ser utilizada la herramienta, pudiéndose introducir y ejecutar especificaciones LOTOS, como se muestra en la siguiente sección.

7.6.6. Ejemplos de ejecución

Para empezar veamos un ejemplo de interacción con la herramienta LOTOS:

```
       LOTOS in Maude version 1.0

Maude> (specification SPEC
type NAT is
  sorts NAT
  opns
    O : -> NAT
    succ : NAT -> NAT
    _+_ : NAT , NAT -> NAT
    eq : NAT , NAT -> Bool
  eqns
    forall N : NAT,
           M : NAT
    ofsort NAT
      O + N = N ;
      succ(N) + M = succ(N + M) ;
    ofsort Bool
      eq(O, O) = true ;
      eq(O, succ(N)) = false ;
      eq(succ(N), O) = false ;
      eq(succ(N), succ(M)) = eq(N,M) ;
endtype
behaviour
 h ! O ; stop
[]
(
   g ! (succ(O)) ; stop
```

```
 |[ g ]|
   g ? x : NAT [ eq(x,succ(0)) ] ; h ! (x + succ(0)) ; stop
)
endspec)

Introduced specification 'SPEC

Maude> (show transitions .)

Trace :(nil).Trace

Condition : true

TRANSITIONS :

1.  {true}{h 0}stop
2.  {x = succ(0)/\ eq(x,succ(0))}{g succ(0)}stop |[g]| h ! succ(succ(0));
    stop

Maude> (cont 2 .)

Trace : g succ(0)

Condition : x = succ(0)/\ eq(x,succ(0))

TRANSITIONS :

1.  {true}{h succ(succ(0))}stop |[g]| stop

Maude> (cont .)

Trace :(g succ(0))(h succ(succ(0)))

Condition : x = succ(0)/\ eq(x,succ(0))

No more transitions .
```

Podemos ver también cómo comprobar que una versión simplificada (finita) de la máquina expendedora de la Sección 4.7 cumple ciertas fórmulas de la lógica modal FULL:

```
Maude> (specification VEN
behaviour
     2p ; big ; collectB ; stop
  []
     1p ; little ; collectL ; stop
endspec)

Introduced specification 'VEN

Maude> (sat ([ big ] ff) /\ ([ little ] ff) .)
```

```
Satisfied .

Maude> (sat [ 2p ] (([ little ] ff) /\ (< big > tt)) .)

Satisfied .

Maude> (sat < collectB > tt .)

Not Satisfied .

Maude> (sat [ 2p ] ([ big ] (< collectB > tt)) .)

Satisfied .

Maude> (show transitions .)

Trace :(nil).Trace

Condition : true

TRANSITIONS :

1.  {true}{2p}big ; collectB ; stop
2.  {true}{1p}little ; collectL ; stop

Maude> (cont 1 .)

Trace : 2p

Condition : true

TRANSITIONS :

1.  {true}{big}collectB ; stop

Maude> (sat < collectB > tt .)

Not Satisfied .

Maude> (cont 1 .)

Trace : 2p big

Condition : true

TRANSITIONS :

1.  {true}{collectB}stop
```

```
Maude> (sat < collectB > tt .)

Satisfied .
```

También hemos utilizado nuestra herramienta para ejecutar ejemplos de mayor entidad, incluyendo el conocido *Alternating Bit Protocol* y el *Sliding Window Protocol* (con más de 550 lineas de código LOTOS) [Tur92]. La herramienta se comporta de forma bastante eficiente, dando respuesta a los comandos introducidos en muy pocos milisegundos.

Mostramos a continuación la especificación utilizada del *Alternating Bit Protocol*:

```
(specification DataLink
   type sequenceNumber is Bool
      sorts
               SeqNum
      opns
               0      :          -> SeqNum
               inc    : SeqNum   -> SeqNum
               _equal_     : SeqNum, SeqNum -> Bool
      eqns forall x, y : SeqNum
           ofsort SeqNum
               inc(inc(x)) = x ;
           ofsort Bool
               x equal x = true ;
               x equal inc(x) = false ;
               inc(x) equal x = false ;
               inc(x) equal inc(y) = (x equal y) ;
   endtype
   type BitString is sequenceNumber
      sorts
               BitString
      opns
               empty  :                    -> BitString
               add    : SeqNum, BitString  -> BitString
   endtype
   type FrameType is Bool
      sorts
               FrameType
      opns
               info, ack  :     -> FrameType
               equal      : FrameType, FrameType -> Bool
      eqns   forall x : FrameType
             ofsort Bool
               equal(info, ack) = false ;
               equal(ack, info) = false ;
               equal(x, x) = true ;
   endtype
behaviour
   hide  tout, send, receive  in
      (  (  transmitter [ get, tout, send, receive ] (0)
```

```
         |||
            receiver [ give, send, receive ] (0)
         )
      |[ tout, send, receive ]|
         line [ tout, send, receive ] )
where
   process transmitter [ get, tout, send, receive ]
                            ( seq : SeqNum) : noexit :=
      get ? data : BitString ;
      sending [ tout, send, receive] (seq, data)
      >> transmitter [ get, tout, send, receive ] (inc(seq))
   endproc
   process sending [ tout, send, receive]
         ( seq : SeqNum, data : BitString ) : exit :=
      send ! info ! seq ! data ;
     (   receive ! ack  ! (inc(seq))  ! empty ; exit
      []  tout ; sending [ tout, send, receive ] (seq, data)
      )
   endproc
   process receiver [ give, send, receive ]
                                ( exp : SeqNum) : noexit :=
      receive ! info ? rec : SeqNum ? data1 : BitString
    ; (   [ rec equal exp ] ->
             give ! data1
           ; send ! ack ! (inc(rec)) ! empty
           ; receiver [ give, send, receive ] (inc(exp))
      []  [ (inc(rec) equal exp) ] ->
             send ! ack ! (inc(rec)) ! empty
           ; receiver [ give, send, receive ] (exp))
   endproc
   process line [ tout, send, receive ] : noexit :=
      send ? f : FrameType ? seq : SeqNum ? data2 : BitString
    ; (   receive ! f ! seq ! data2
        ; line [ tout, send, receive ]
      []  i
        ; tout
        ; line [ tout, send, receive ]
      )
   endproc
endspec)
```

7.7. Comparación con otras herramientas

En esta sección compararemos la implementación presentada en este capítulo con la implementación alternativa que hemos realizado siguiendo el enfoque de reglas de inferencia como reescrituras, y con otras herramientas para LOTOS no basadas en lógica de reescritura.

7.7.1. LOTOS con el enfoque de reglas de inferencia como reescrituras

Como dijimos en la introducción a este capítulo, hemos implementado también una herramienta similar a la descrita aquí utilizando el enfoque alternativo en el que las transiciones se representan mediante términos y las reglas de inferencia mediante reglas de reescritura. Esta implementación se presenta con todo detalle en [Ver02a]. Aquí comentaremos primero las diferencias entre ella y la implementación de CCS presentada en el Capítulo 4, para después comparar las dos implementaciones de LOTOS.

Ya que la semántica de LOTOS es mucho más complicada que la de CCS, y las especificaciones LOTOS son bastante más complicadas que los simples procesos CCS, a la hora de implementar la semántica de LOTOS tuvimos que mejorar las ideas presentadas en el Capítulo 4 en diversas direcciones.

En primer lugar, para presentar las reglas semánticas de una forma más general y elegante, escribimos las reglas para cada operador LOTOS asumiendo la presencia de metavariables en cualquier sitio donde pueden aparecer estas dentro de una transición, en vez de como lo hicimos en el Capítulo 4, donde se escribieron reglas diferentes dependiendo de donde aparecieran las metavariables. Añadimos a continuación otras reglas adicionales que reducen los juicios sin metavariables en todos los sitios a los anteriores. Si hubiéramos hecho esa simplificación en el caso de CCS, para el operador de prefijo solo tendríamos la regla

```
rl [pref] :   A . P -- ?A -> ?P
        => ----------------------
             [?A := A] [?P := P] .
```

Pero en cambio tendríamos las siguientes reglas *generales* que permiten otras clases de juicios como "conclusión":

```
rl [meta] :   P -- A -> P'
        => ----------------------------
              P -- ?(NEW1)A -> ?(NEW2)P
              [ ?(NEW1)A == A ]
              [ ?(NEW2)P == P' ] .

rl [meta] :   P -- ?A -> P'
        => ----------------------
              P -- ?A -> ?(NEW2)P
              [ ?(NEW2)P == P' ] .

rl [meta] :   P -- A -> ?P
        => ----------------------
              P -- ?(NEW1)A -> ?P
              [ ?(NEW1)A == A ] .
```

Continuando con la idea de presentar la semántica de una forma lo más general posible, utilizamos también un operador `_[[_/_]]` para representar la sustitución sintáctica de metavariables por sus valores concretos, en lugar de las operaciones `<act_:=_>_` utilizadas

para CCS, cuando estos se propagan al resto de juicios. La mejora está en que podemos definir el comportamiento de este nuevo operador al metanivel *solo una vez*, en vez de tener que definir las diversas operaciones auxiliares sobrecargadas utilizadas en el Capítulo 4.

En cuanto a la eficiencia, cambiamos los conjuntos de juicios por *secuencias* de juicios en los que el orden importa, para evitar los múltiples encajes de patrones módulo conmutatividad. A partir de aquí los juicios se mantienen ordenados y se prueban de izquierda a derecha. También modificamos la estrategia de búsqueda para que solo se intente reescribir el primer juicio de la secuencia. Ya que la estrategia tiene que comprobar que todos los juicios se reducen al conjunto vacío, si el primer juicio no puede ser reescrito, ya no es necesario intentar reescribir el resto de juicios, pues es ya evidente que por debajo del nodo actual en el árbol de reescrituras no habrá ningún nodo que represente la secuencia vacía. Sin embargo, este cambio puede afectar a la forma en que las premisas se escriben en la regla semántica, ya que para reducir un juicio no deberían hacer falta ligaduras producidas por un juicio posterior (a la derecha).

Esta segunda implementación de LOTOS se integró con Full Maude (el cual es altamente extensible [Dur99]) añadiendo la capacidad de introducir especificaciones LOTOS a Full Maude y la de trabajar con ellas por medio de nuevos comandos específicos, como también hemos hecho en este capítulo. También se permiten especificaciones ACT ONE que se traducen automáticamente a módulos funcionales que se integran con la semántica simbólica.

La diferencia más importante entre las dos implementaciones, aparte de la representación de la semántica, que es bastante diferente, se encuentra en las cosas que se hacen (o pueden hacerse) al nivel objeto (el nivel de la representación de la semántica) y el metanivel (utilizando reflexión). En [Ver02a], se define una operación de búsqueda al metanivel, para comprobar si una transición LOTOS es posible. La operación recorre el árbol con todas las posibles reescrituras de un término, moviéndose continuamente entre el nivel objeto y el metanivel. En la implementación descrita en este capítulo la búsqueda se realiza siempre al nivel objeto, lo cual la hace bastante más rápida y simple. Cierto es que el hecho de estar moviéndose continuamente entre los dos niveles nos permite definir en [Ver02a] más cosas al metanivel, como la operación de sustitución sintáctica o la extracción de variables, definida utilizando valores del tipo `Term`. Aquí no podíamos seguir esa solución ya que la búsqueda ocurre completamente en el nivel objeto, por lo que hemos seguido un enfoque diferente, como hemos explicado en la Sección 7.5.1, en el que los módulos traducción de una especificación en ACT ONE se extienden de forma automática, gracias a las capacidades de metaprogramación de Maude, con ecuaciones que definen (al nivel objeto) estas operaciones sobre expresiones con la nueva sintaxis. Este enfoque nos parece mucho más elegante y general.

7.7.2. Otras herramientas

La herramienta *Concurrency Workbench of the New Century* (CWB-NC) es una herramienta de verificación automática con la que pueden ejecutarse y analizarse sistemas dados en diferentes lenguajes de especificación [CS02]. En lo que se refiere a LOTOS, CWB-NC acepta *Basic LOTOS*, ya que la herramienta no soporta álgebras de procesos con paso de

valores. El diseño del sistema hace énfasis en la independencia del lenguaje de sus rutinas de análisis, localizando los procedimientos específicos de cada lenguaje, lo que permite al usuario cambiar el lenguaje de descripción de sistemas utilizando la herramienta *Process Algebra Compiler* [CMS95], que traduce las definiciones de la semántica operacional de un lenguaje concreto en código en SML. Nosotros hemos seguido un enfoque similar, separando la semántica del lenguaje de los procesos de análisis que la utilizan, aunque hemos intentado mantener la representación de la semántica en un nivel tan alto como sea posible, sin que deje de ser ejecutable. De esta forma evitamos tener que traducir las representaciones semánticas a otros lenguajes.

El conjunto de herramientas *Caesar/Aldebaran Development Package* (CADP) se utiliza para el desarrollo de protocolos, con diversas funcionalidades, desde la simulación interactiva hasta la verificación formal [JHA+96]. Para poder dar soporte a diferentes lenguajes de especificación, CADP utiliza representaciones internas de bajo nivel, lo cual obliga al implementador de una nueva semántica a construir compiladores que generen estas representaciones. CADP ya ha sido utilizado para implementar FULL [BS01], aunque con la fuerte restricción a tipos finitos y con la semántica estándar de LOTOS, en vez de con la semántica simbólica en la que se basa FULL.

7.8. Conclusiones

Hemos presentado un nuevo ejemplo de cómo se pueden utilizar la lógica de reescritura y Maude como marco semántico y como metalenguaje, con el que se pueden construir entornos y herramientas completas para ejecutar lenguajes de especificación formal. En este proceso la reflexión juega un papel decisivo.

Hemos implementado la semántica simbólica de LOTOS en Maude siguiendo el enfoque de transiciones como reescrituras, ya utilizado para CCS en el Capítulo 5. Aunque LOTOS es un lenguaje bastante más complejo que CCS, en lo que tiene que ver estrictamente con los procesos hemos podido utilizar las mismas técnicas presentadas para este. Uno de los aspectos más novedosos de LOTOS es la inclusión de tipos de datos, con una sintaxis y semántica definida por el usuario. Hemos implementado una traducción de especificaciones en ACT ONE a módulos funcionales en Maude, la cual permite ejecutar estas especificaciones en Maude, utilizando su máquina de reducción, cuyo rendimiento, en lo que se refiere a tiempo de ejecución, es muy alto. Estos módulos funcionales se integran con la semántica, obteniéndose así una implementación de la semántica simbólica de LOTOS con tipos de datos definidos por el usuario.

El enfoque de transiciones como reescrituras es diferente al utilizado en el Capítulo 4 para CCS y en [Ver02a] para LOTOS, y presenta diversas ventajas, como se ha explicado en la Sección 7.7.1.

Finalmente, hemos implementado en Maude un interfaz de usuario para nuestra herramienta, que permite que el usuario no utilice Maude directamente, sino que utilice una herramienta escrita sobre Maude cuya entrada es una especificación LOTOS, y donde esta especificación puede ser ejecutada mediante comandos que recorren el correspondiente sistema de transiciones etiquetado.

Capítulo 8

Protocolo de elección de líder de IEEE 1394

Durante los últimos años se ha venido utilizando la lógica de reescritura y Maude en la especificación de sistemas reales, principalmente en aplicaciones de arquitecturas distribuidas [DMT00b, DV01, ADV01c, ADV01a, ADV01b] y protocolos de comunicación [DMT98, DMT00a]. En [DMT98] se presenta una metodología formal para especificar y analizar protocolos de comunicación (que ya presentamos en la introducción de esta tesis, página 5), estructurada como una secuencia de métodos incrementalmente más potentes, que incluye los pasos siguientes:

1. *Especificación formal*, para obtener un modelo formal del sistema, en el que se hayan resuelto las posibles ambigüedades.

2. *Ejecución de la especificación*, con el propósito de poder simular y depurar la especificación, conduciendo a mejores especificaciones.

3. *Análisis formal mediante "model-checking"*, con el fin de encontrar errores a base de considerar todos los posibles comportamientos de un sistema altamente distribuido y no determinista.

4. *Análisis por "narrowing"*, en el cual se analizan todos los comportamientos de un conjunto posiblemente infinito de estados descrito por una expresión simbólica.

5. *Demostración formal*, donde se verifica la corrección de propiedades críticas por medio de alguna técnica formal.

En este capítulo utilizamos los métodos 1, 2 y 3 para especificar y analizar tres descripciones, a diferentes niveles de abstracción, del protocolo de elección de líder dentro de la especificación del bus multimedia en serie IEEE 1394 (conocido como *FireWire*), y daremos indicaciones de cómo utilizar el método 5 para verificar las especificaciones. No hemos podido realizar ningún análisis por *narrowing* ya que el sistema Maude no lo soporta (al menos al nivel objeto).

Aunque para desarrollar el estándar IEEE 1394 no se utilizaron métodos formales, varios aspectos del sistema han sido descritos utilizando gran variedad de técnicas diferentes, entre las que se incluye los autómatas I/O [DGRV97, Rom01], μCRL [SZ98] y E-LOTOS [SM98, SV01]. De esta forma este ejemplo se está convirtiendo en un caso de estudio típico para la comparación de métodos formales [MS00, SMR01]. En este capítulo mostramos cómo Maude también puede ser utilizado como lenguaje de especificación formal. Al efecto se utiliza el estilo de especificación orientado a objetos de Maude (véase la Sección 2.3), que permite la formalización de sistemas de objetos concurrentes, tanto síncronos como asíncronos. En particular, se estudian los aspectos del protocolo relacionados con el tiempo, por lo que añadiremos conceptos temporales a estas especificaciones, siguiendo ideas presentadas en [ÖM02]. La corrección del protocolo se ha probado de dos formas distintas. Primero, las descripciones se han validado por medio de una exploración exhaustiva de todos los posibles comportamientos y estados alcanzables a partir de una configuración inicial (arbitraria) de la red, comprobando que siempre ocurre que se elige un único líder. Veremos cómo realizar este análisis en la Sección 8.4. En segundo lugar, se ha comprobado la corrección del protocolo mediante una demostración formal que prueba que toda red conexa y acíclica cumple las propiedades que se desean. Este trabajo es parte de la tesis doctoral de Isabel Pita [Pit03]. En la Sección 8.5 resumiremos las principales ideas.

La mayoría de los trabajos presentados en [SMR01] utilizan formalismos abstractos y herramientas que no son apropiadas para la ejecución directa del protocolo, aunque de una u otra forma cada enfoque aporta una visión útil del tema. Pero sin duda los métodos de especificación ejecutables y sus herramientas facilitan una representación más completa. Como se apunta en [Ölv00], la lógica de reescritura y Maude se sitúan a un *nivel operacional intermedio*, que puede ayudar considerablemente a llenar el hueco entre las especificaciones más abstractas, orientadas a las propiedades, y las implementaciones concretas, proporcionando al tiempo soporte a especificaciones ejecutables, técnicas útiles para el razonamiento formal automático o semi-automático (como el análisis formal basado en estrategias presentado en la Sección 8.4) y un modelo matemático preciso del sistema.

Parte del trabajo contenido en este capítulo se presentó por primera vez en el *Third International Workshop on Rewriting Logic and its Applications, WRLA 2000*, y fue publicado en [VPMO00]. También se presentó en el *International Workshop on Application of Formal Methods to IEEE 1394 Standard* [VPMO01b, VPMO01a] en 2001, evento que ha dado lugar a un volumen especial de la revista *Formal Aspects of Computing* donde se ha incluido una versión extendida de nuestro trabajo [VPMO02].

Nuestro trabajo relacionado con el protocolo de elección de líder dentro del estándar IEEE 1394 comenzó antes [SV99a], estudiando las cualidades del álgebra de procesos E-LOTOS [ISO01] para describir este tipo de protocolos. Una versión extendida y mejorada de este trabajo se ha publicado en la revista *Computer Networks* [SV01]. En la Sección 8.8 realizaremos una comparación entre el trabajo presentado en este capítulo y el trabajo con E-LOTOS.

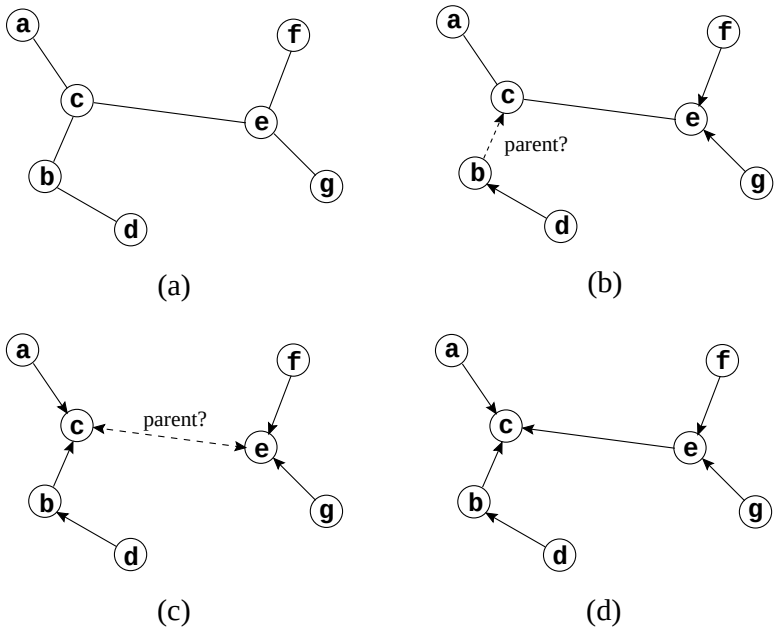


Figura 8.1: Configuraciones de la red durante el protocolo de elección de líder.

8.1. Descripción informal del protocolo

El bus multimedia en serie IEEE 1394 [IEE95] conecta sistemas y dispositivos para que transporten cualquier forma de vídeo y audio digital de forma rápida, fiable y barata. Su arquitectura es escalable, y sigue el enfoque "conectar sin apagar" (*hot-pluggable*), es decir, un diseñador o usuario puede añadir o eliminar sistemas y periféricos de forma fácil en cualquier momento. El estándar IEEE 1394 completo es complejo y está formado por varios subprotocolos diferentes, cada uno encargado de una tarea distinta, como transferencia de datos entre nodos de la red, arbitraje del bus, elección de líder, etc. El estándar se describe en capas, al estilo de OSI (Open Systems Interconnection), y cada capa se divide a su vez en diferentes fases.

En este capítulo nos limitaremos a describir la fase conocida como *identificación del árbol* de la capa física. De manera informal, la fase de identificación del árbol del estándar IEEE 1394 consiste en un protocolo de elección de líder que tiene lugar tras una inicialización del bus en la red, es decir, cuando se añade o se elimina un nodo de la red. Inmediatamente después de la inicialización de la red, todos los nodos tienen el mismo estado, y solo saben a qué otros nodos están conectados. Debemos elegir un líder (o raíz) para que sirva de autoridad en el bus en las siguientes fases del estándar IEEE 1394. La Figura 8.1(a) muestra el estado inicial de una posible red. Las conexiones entre los nodos se indican mediante líneas continuas. El protocolo tiene éxito cuando la red original es conexa y acíclica, aunque el protocolo es capaz de detectar ciclos y avisar del error.

Cada nodo lleva a cabo una serie de negociaciones con sus vecinos para establecer el sentido de la relación padre-hijo entre ellos. De forma más precisa, si un nodo tiene n

conexiones, empezará recibiendo peticiones "sé mi padre" desde todas ellas, o desde todas menos una. Una vez que se han hecho n o $n-1$ peticiones de este estilo, el nodo pasa a una fase de reconocimiento, en la que envía mensajes "tú eres mi hijo" a todos los nodos que enviaron peticiones "sé mi padre" en la fase previa. Cuando se han enviado todos los reconocimientos, puede ocurrir que el nodo tenga n hijos y, por tanto, sea el nodo raíz, o que tenga $n-1$ hijos, en cuyo caso el nodo envía una petición "sé mi padre" por la conexión aún no utilizada y espera el reconocimiento del nodo al otro lado de esta conexión. Las hojas se saltan la fase inicial de recepción de peticiones y pasan directamente a este punto; como ellas solo tienen una conexión conocen por defecto la forma de llegar al padre. La Figura 8.1(b) muestra el instante en el que los nodos `d`, `f` y `g` saben ya quién es su padre (conexiones continuas con la flecha apuntando al padre), y el nodo `b` está pidiendo al nodo `c` que sea su padre (la relación se muestra con una línea discontinua).

La comunicación entre nodos es asíncrona, por lo que es posible que dos nodos se pidan simultáneamente el uno al otro que sea su padre, produciéndose un *conflicto de raíz* (véase Figura 8.1(c)). Para resolver el conflicto, cada nodo selecciona un valor booleano aleatorio, y dependiendo del mismo se espera un periodo corto o largo antes de volver a enviar la petición "sé mi padre". Evidentemente, esto puede mantener el conflicto, pero la hipótesis de equidad garantiza que más tarde o más temprano uno se convertirá en la raíz del árbol.

Cuando todas las negociaciones han concluido, el nodo que se ha hecho padre de *todos* los nodos a los que está conectado será el nodo raíz de un árbol de recubrimiento de la red. Véase la Figura 8.1(d), en la cual el nodo `c` es el nodo raíz.

En las secciones siguientes se presentan tres descripciones de este protocolo a diferentes niveles de abstracción.

8.2. Descripción del protocolo con comunicación síncrona

Empezamos con una descripción simple del protocolo, sin consideraciones en lo que se refiere al tiempo, en la que la comunicación entre nodos se supone que es síncrona, de modo que todo mensaje se envía y se recibe simultáneamente. En consecuencia, no hay necesidad de mensajes de reconocimiento, y no puede haber conflictos.

En el estándar IEEE 1394 se habla de nodos y comunicaciones entre ellos, que equivalen de forma natural a objetos y mensajes entre ellos. En esta primera descripción los nodos están representados por objetos de una clase `Node` con los siguientes atributos:

- `neig : SetIden`, el conjunto de identificadores de nodos vecinos con los que el nodo todavía no se ha comunicado. Este atributo se inicializará con el conjunto de todos los nodos (identificadores de objeto) conectados a este, y se irá reduciendo a medida que haya comunicaciones con otros nodos que quieran que este sea su padre, hasta que se haga vacío, con lo que el nodo será la raíz, o solo contenga un elemento, que sería el padre del nodo; y
- `done : Bool`, una etiqueta que se hará cierta cuando la fase de identificación del árbol para este nodo haya terminado, bien porque haya sido elegido como nodo raíz, o bien porque ya sepa qué nodo es su padre.

Ya que la comunicación es síncrona, no son necesarios mensajes de petición "sé mi padre" ni mensajes de reconocimiento. Sin embargo, sí añadimos un mensaje "líder", que será enviado por el nodo elegido como raíz para indicar que ha sido elegido un líder. Esto nos proporciona un método con el cual comprobar la propiedad de que se elije un único líder eventualmente (véase Sección 8.4).

En el siguiente módulo se introducen los identificadores de nodos y los conjuntos de identificadores. Un identificador es un conjunto de identificadores unitario y los conjuntos mayores se construyen por yuxtaposición (`__`). Este operador de unión se declara como asociativo, conmutativo, y con el conjunto vacío como elemento identidad. El encaje de patrones se realizará módulo estos atributos, de forma que un patrón como `J NEs` representará cualquier conjunto que tenga un elemento `J` y un resto `NEs`.

```
(fmod IDENTIFIERS is
   protecting QID .

   sorts Iden SetIden .
   subsorts Qid < Iden < SetIden .

   op empty : -> SetIden .
   op __ : SetIden SetIden -> SetIden [assoc comm id: empty] .

   var S : SetIden .
   eq S S = S .
endfm)
```

El módulo orientado a objetos en el que se describe el protocolo comienza declarando los identificadores de nodos como identificadores de objetos válidos, la clase `Node` con sus atributos y el mensaje `leader`, que incluye como parámetro el identificador del nodo elegido como líder:

```
(omod FIREWIRE-SYNC is
   protecting IDENTIFIERS .
   subsort Iden < Oid .

   class Node | neig : SetIden, done : Bool .
   msg leader_ : Iden -> Msg .
```

Ahora hay que especificar el comportamiento de los nodos mediante reglas de reescritura. En este caso nos basta con dos. La primera regla describe cómo un nodo `J`, que solo contenga un identificador `I` en su atributo `neig`, envía una petición "sé mi padre" al nodo `I`, y cómo este nodo recibe la petición y elimina el nodo `J` de su conjunto de comunicaciones pendientes; con ello el nodo `J` también finaliza su fase de identificación, poniendo a cierto el atributo `done`. Hablamos de enviar y recibir una petición de forma figurada, pues al ser la regla `rec` una regla síncrona, no hay ni envío ni recepción de mensajes de forma explícita, pero sí hay un intercambio implícito de información, en el cual un nodo actúa como emisor y el otro como receptor.

```
vars I J : Iden .
var NEs : SetIden .

rl [rec] :
   < I : Node | neig : J NEs, done : false >
   < J : Node | neig : I, done : false >
=> < I : Node | neig : NEs >  < J : Node | done : true > .
```

Nótese que el no determinismo aparece cuando hay dos nodos conectados con un único identificador en sus atributos `neig`. En este caso cualquiera de ellos tiene la posibilidad de actuar como emisor.

La siguiente regla establece cuándo un nodo es elegido como líder.

```
rl [leader] :
   < I : Node | neig : empty, done : false >
=> < I : Node | done : true >  (leader I) .
endom)
```

En la Sección 8.5 mostraremos cómo abordar la demostración de corrección de esta descripción síncrona del protocolo.

8.3. Descripción con comunicación asíncrona con tiempo

La descripción anterior es muy simple, pero no da una visión fidedigna de los eventos que ocurren en el protocolo real, en el que los mensajes se envían a través de cables de longitud variable, con lo que el paso de mensajes es asíncrono y está sujeto a retrasos. Al ser la comunicación asíncrona, se hacen necesarios los mensajes de reconocimiento, y puede aparecer el problema de que dos nodos puedan simultáneamente pedir uno al otro que sea su padre, produciéndose un conflicto de raíz. Si utilizáramos la comunicación asíncrona explícita a través de mensajes de Maude, llegaríamos a una descripción del protocolo que no funcionaría de la forma esperada, pues sería posible que la fase de conflicto y la fase de recepción de peticiones "sé mi padre" se alternaran de forma indefinida. En consecuencia, no debemos ignorar los aspectos relacionados con el tiempo del protocolo, y en la fase de conflicto de raíz los nodos tienen que esperar un periodo de tiempo corto o largo (elegido aleatoriamente) antes de enviar de nuevo la petición "sé mi padre". Otros aspectos relacionados con el tiempo, como son la detección de ciclos o la presencia en un nodo de un parámetro que le fuerza a permanecer más tiempo en la fase de recepción de peticiones por parte de los hijos, se tratarán en la Sección 8.3.3.

Hay que precisar que no representamos todos los detalles concretos del protocolo tal y como se describen en [IEE95], dando descripciones más abstractas. En particular, representamos la comunicación entre nodos por medio de mensajes discretos, en vez de la colocación y eliminación de señales continuas sobre los cables utilizadas en el estándar. La razón es el nivel de abstracción que queremos mantener, y no la incapacidad de nuestro enfoque para representar tales aspectos. Así, por ejemplo, podríamos haber utilizado objetos para representar los cables con atributos que tuvieran el valor actual de la señal. En ese caso, los nodos detectarían el estado del cable examinando el correspondiente atributo.

Antes de presentar la nueva descripción del protocolo con tiempo, describimos de forma breve las ideas presentadas en [Ölv00, ÖM02] sobre cómo introducir conceptos de tiempo en la lógica de reescritura y Maude, y de forma particular en una especificación orientada a objetos. En nuestra opinión, la introducción de aspectos temporales en una especificación, tal y como se realiza a continuación, es bastante natural y modular.

8.3.1. El tiempo en la lógica de reescritura y Maude

Una teoría de reescritura con tiempo real es una teoría de reescritura con un tipo de datos `Time` que representa los valores de tiempo y que cumple ciertas propiedades, como ser un monoide conmutativo $(\texttt{Time}, +, 0)$ con operaciones $\leq$, $<$ y $\dot{-}$ ("monus"). Utilizamos el módulo `TIME-DOMAIN` para representar los valores de tiempo, con un tipo `Time` cuyos valores son los números naturales que es subtipo del tipo `TimeInf`, que contiene además la constante `INF` que representa $+\infty$ (véase [Ölv00]).

```
(fmod NAT-INF is
  protecting MACHINE-INT .

  sorts Nat NzNat NatInf .

  subsort Nat < MachineInt .
  subsort NzNat < Nat .
  subsort NzNat < NzMachineInt .
  subsort Nat < NatInf .

  var NZ : NzMachineInt .

  mb 0 : Nat .
  cmb NZ  : NzNat if NZ > 0 .

  op INF : -> NatInf .

  op _plus_ : NatInf NatInf -> NatInf [assoc comm] .
  op _monus_ : NatInf Nat -> NatInf .

  op _lt_ : NatInf NatInf -> Bool .
  op _le_ : NatInf NatInf -> Bool .
  op min : NatInf NatInf -> NatInf [comm] .
  op max : NatInf NatInf -> NatInf [comm] .


  vars N N' : Nat .
  vars NI NI' : NatInf .

  eq N plus N' = N + N' .
  eq NI plus INF = INF .

  eq N monus N' = if N lt N' then 0 else (N - N') fi .
  eq INF monus N' = INF .
```

```
  eq N lt N' = N < N' .
  eq N lt INF = true .
  eq INF lt NI = false .

  eq N le N' = N <= N' .
  eq NI le INF = true .
  eq INF le N = false .

  ceq min(NI, NI') = NI if NI le NI' .
  ceq max(NI, NI') = NI' if NI le NI' .

endfm)

(fmod TIME-DOMAIN is
  protecting NAT-INF .

  sorts Time TimeInf .
  subsorts Nat < Time < TimeInf .
  subsort NatInf < TimeInf .
endfm)
```

Las reglas se dividen en *reglas tick*, que modelan el paso del tiempo en un sistema, y *reglas instantáneas* que modelan cambios en partes del sistema, y que se supone tardan tiempo cero. Para asegurar que el tiempo avanza uniformemente en todas las partes de un estado, se hace necesario un nuevo tipo de datos `ClockedSystem`, con un constructor libre

```
{_|_} : State Time -> ClockedSystem.
```

En el término $\{\ s\ \mid\ t\ \}$, s denota el estado *global* del sistema y t denota el tiempo transcurrido en un cómputo si el estado inicial del reloj tenía el valor 0. El paso uniforme del tiempo se asegura entonces si cada regla tick es de la forma

$$\{\ s\ \mid\ t\ \} \longrightarrow \{\ s'\ \mid\ t+\tau\ \},$$

donde τ representa la duración de la regla. Estas reglas se denominan *reglas globales* porque reescriben términos de tipo `ClockedSystem`. El resto de reglas se denominan *reglas locales*, porque no actúan sobre el sistema completo, sino solo sobre algunas de sus componentes. Al manejar reglas locales se permite el paralelismo, ya que varias de ellas pueden aplicarse al mismo tiempo en diferentes partes del sistema. Las reglas locales se consideran siempre como reglas instantáneas que tardan un tiempo cero.

En general, también debe asegurarse que el tiempo no transcurre si hay acciones instantáneas que realizar. Aunque en muchos casos esto se puede lograr añadiendo condiciones a las reglas tick, de tal forma que el tiempo no transcurra si hay alguna regla de tiempo crítico que se pueda ejecutar (y esto es lo que ocurre en nuestro caso, como veremos más adelante), un enfoque más general consiste en dividir las reglas de una teoría de reescritura con tiempo real en reglas *impacientes* y reglas *perezosas*, y utilizar estrategias internas que

restrinjan las reescrituras posibles, haciendo que la aplicación de reglas impacientes tenga precedencia sobre la de reglas perezosas (véase la Sección 8.4.1).

Estas ideas pueden aplicarse de igual forma a sistemas orientados a objetos [ÖM02]. En tal caso, el estado global será un término del tipo `Configuration`, y ya que este tiene una estructura muy rica, es necesario disponer de una operación explícita δ que denote el efecto del paso del tiempo en el estado completo. De esta manera, la operación δ será definida para cada posible elemento de una configuración de objetos y mensajes, y habrá ecuaciones que distribuyan el efecto del tiempo al sistema completo. En este caso, las reglas tick serán de la forma

$$\{\ s\ \mid\ t\ \} \longrightarrow \{\ \delta(s,\tau)\ \mid\ t+\tau\ \}.$$

También es útil una operación `mte` que devuelva el máximo periodo de tiempo que puede transcurrir antes de que tengan que ejecutarse acciones de tiempo crítico, y que será definida de forma separada para cada objeto y mensaje. El módulo general presentado a continuación declara estas operaciones y cómo se distribuyen sobre los elementos (recuérdese que `none` es la configuración vacía):

```
(omod TIMED-OO-SYSTEM is
   protecting TIME-DOMAIN .
   sorts State ClockedSystem .
   subsort Configuration < State .

   op `{_|_`} : State Time -> ClockedSystem .
   op delta : Configuration Time -> Configuration .
   op mte : Configuration -> TimeInf .

   vars CF CF' : Configuration .
   var T : Time .

   eq delta(none, T) = none .
   ceq delta(CF CF', T) = delta(CF, T) delta(CF', T)
                          if CF =/= none and CF' =/= none .

   eq mte(none) = INF .
   ceq mte(CF CF') = min(mte(CF), mte(CF'))
                     if CF =/= none and CF' =/= none .
endom)
```

8.3.2. Segunda descripción del protocolo

En esta segunda descripción del protocolo, cada nodo pasa a través de una serie de fases diferentes (como se explicó en la Sección 8.1) que se declaran como sigue:

```
(fmod PHASES is
    sort Phase .
    ops rec ack waitParent contention self : -> Phase .
endfm)
```

Cuando un nodo se encuentra en la fase `rec` está recibiendo peticiones "sé mi padre" de sus vecinos. En la fase `ack` el nodo envía mensajes de reconocimiento "tú eres mi hijo" a todos los nodos que enviaron "sé mi padre" en la fase previa. En la fase `waitParent` el nodo espera el mensaje de reconocimiento de su padre. En la fase `contention` el nodo elige si esperará un periodo de tiempo corto o largo antes de enviar de nuevo el mensaje "sé mi padre". Por último, un nodo se encontrará en la fase `self` cuando ha sido elegido como líder o cuando ya ha recibido el reconocimiento de su padre; en ambos casos el protocolo de elección de líder ha terminado para él.

Los atributos de la clase `Node`, definida en el módulo `FIREWIRE-ASYNC` que extiende al módulo `TIMED-OO-SYSTEM`, son ahora los siguientes:

```
class Node | neig : SetIden, children : SetIden,
             phase : Phase, rootConDelay : DefTime .
```

El atributo `children` representa el conjunto de hijos que tienen que ser reconocidos, `phase` representa la fase en la que se encuentra el nodo, y `rootConDelay` es un *reloj* utilizado en la fase de conflicto de raíz. El tipo `DefTime` extiende al tipo `Time`, que representa los valores del tiempo, con una nueva constante `noTimeValue` utilizada cuando el reloj está parado [ÖM02].

```
sort DefTime .
subsort Time < DefTime .
op noTimeValue : -> DefTime .
```

Además del mensaje `leader`, introducimos dos nuevos mensajes que tienen como argumentos el emisor, el receptor y el tiempo necesario para alcanzar al receptor:

```
msg from_to_be‘my‘parent‘with‘delay_ : Iden Iden Time -> Msg .
msg from_to_acknowledgement‘with‘delay_ : Iden Iden Time -> Msg .
```

Por ejemplo, el mensaje `from I to J be my parent with delay T` denota que una petición "sé mi padre" ha sido enviada desde el nodo `I` al nodo `J`, y que llegará a `J` dentro de `T` unidades de tiempo. Un mensaje con tiempo 0 es *urgente*, en el sentido de que tiene que ser atendido por el receptor antes de que pase ningún tiempo. La operación `mte` asegurará que esta restricción se cumple.

La primera regla[1] establece que un nodo `I` en la fase `rec`, y con más de un vecino, puede recibir una petición "sé mi padre" que tenga tiempo 0 de uno de sus vecinos `J`. El identificador `J` se almacena en el atributo `children`:

```
vars I J K : Iden .
vars NEs CHs : SetIden .
```

[1] Aunque para simplificar la explicación aquí presentamos las reglas como locales reescribiendo términos de tipo `Configuration`, en la especificación completa utilizamos reglas globales que reescriben términos de tipo `ClockedSystem`, para evitar problemas con la función `mte`, que tiene un argumento de tipo `Configuration`.

```
crl [rec] :
    (from J to I be my parent with delay 0)
    < I : Node | neig : J NEs, children : CHs, phase : rec >
=>  < I : Node | neig : NEs, children : J CHs >
   if NEs =/= empty .
```

Cuando un nodo se encuentra en la fase `rec` y solo tiene una conexión no utilizada, puede pasar a la siguiente fase, `ack`, o puede recibir la última petición antes de entrar en dicha fase:

```
rl [recN-1] :
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | phase : ack >   .

rl [recLeader] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | neig : empty, children : J CHs, phase : ack > .
```

Obsérvese que la regla `recN-1` podría aplicarse aunque hubiera un mensaje "sé mi padre". Esto ocasionaría un conflicto de raíz, como veremos más adelante.

En la fase de reconocimiento el nodo envía los reconocimientos "tú eres mi hijo" a todos los nodos que anteriormente le habían enviado peticiones "sé mi padre", cuyos identificadores, por tanto, se encuentran en el atributo `children`:

```
rl [ack] :
   < I : Node | children : J CHs, phase : ack >
=> < I : Node | children : CHs >
   (from I to J acknowledgement with delay timeLink(I,J)) .
```

La operación

```
op timeLink : Iden Iden -> Time .
```

representa una tabla con los valores del tiempo que tarda un mensaje en llegar de un nodo a otro (véase un ejemplo en la Sección 8.3.4).

Cuando se han enviado todos los mensajes de reconocimiento, o bien el nodo tiene el atributo `neig` vacío y, por tanto, es el nodo raíz, o bien envía una petición "sé mi padre" por la única conexión no utilizada y espera un reconocimiento de su padre. Nótese de nuevo que las hojas se saltan la fase inicial de recepción de peticiones y pasan directamente a este punto.

```
rl [ackLeader] :
   < I : Node | neig : empty, children : empty, phase : ack >
=> < I : Node | phase : self >  (leader I) .
```

```
rl [ackParent] :
   < I : Node | neig : J, children : empty, phase : ack >
=> < I : Node | phase : waitParent >
   (from I to J be my parent with delay timeLink(I,J)) .

rl [wait1] :
   (from J to I acknowledgement with delay 0)
   < I : Node | neig : J, phase : waitParent >
=> < I : Node | phase : self >  .
```

Habiendo modelado las comunicaciones por medio de mensajes, y considerando un medio libre de fallos en el que los mensajes no se "pierden", no es necesaria la confirmación por parte de los hijos reconocidos. Esto no es así en el estándar IEEE 1394, donde el padre espera reconocimientos de todos sus hijos antes de enviar su propia petición "sé mi padre".

Una vez enviada una petición al padre, el nodo espera el reconocimiento. Si en vez de un reconocimiento llega una petición "sé mi padre", entonces este nodo y el que originó la petición entran en conflicto para ver quién es finalmente el líder.

En el estándar IEEE 1394, el conflicto se resuelve eligiendo de forma aleatoria un valor booleano `b` y esperando un periodo de tiempo, corto o largo dependiendo del valor de `b`, antes de mirar por el puerto correspondiente si ha llegado una petición "sé mi padre" desde el otro nodo. Si nos encontramos con una petición, este nodo pasa a ser la raíz y envía un reconocimiento al otro; si el mensaje no ha llegado, entonces el nodo enviará de nuevo la petición "sé mi padre".

En nuestra representación se elige el booleano utilizando un generador de números aleatorios, y se espera el tiempo correspondiente. Se utilizan las constantes `ROOT-CONT-FAST` y `ROOT-CONT-SLOW`, definidas en el estándar, para representar un tiempo corto y largo, respectivamente. Si durante la espera llega una petición "sé mi padre", la espera se aborta y la petición es atendida; si el tiempo de espera expira, entonces el nodo reenvía la petición "sé mi padre":

```
rl [wait2] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, phase : waitParent >
   < RAN : RandomNGen | seed : N >
=> < I : Node | phase : contention,
                rootConDelay : if (N % 2 == 0) then ROOT-CONT-FAST
                               else ROOT-CONT-SLOW fi >
   < RAN : RandomNGen | seed : random(N) > .

rl [contenReceive] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, phase : contention >
=> < I : Node | neig : empty, children : J, phase : ack,
                rootConDelay : noTimeValue > .

rl [contenSend] :
   < I : Node | neig : J, phase : contention, rootConDelay : 0 >
```

```
=> < I : Node | phase : waitParent, rootConDelay : noTimeValue >
   (from I to J be my parent with delay timeLink(I,J)) .
```

Los objetos de la clase `RandomNGen` son generadores de números pseudoaleatorios. La declaración de la clase y la función `random` son las siguientes:

```
class RandomNGen | seed : MachineInt .

op random : MachineInt -> MachineInt .   *** next random number

var N : MachineInt .
eq random(N) = ((104 * N) + 7921) % 10609 .
```

La corrección de esta elección aleatoria se trata en [SV99b], donde se demuestra que el algoritmo de resolución del conflicto de raíz garantiza su terminación de forma satisfactoria eventualmente (con probabilidad uno). Este resultado se extiende en [FS02], donde se deriva una relación entre el número de intentos de resolver el conflicto y la probabilidad de una salida satisfactoria.

Ahora tenemos que definir cómo afecta el tiempo a los objetos y a los mensajes, es decir, tenemos que definir la operación `delta` que denota el efecto del paso del tiempo en objetos y mensajes, y también cuál es el máximo transcurso de tiempo permitido, para asegurar que las acciones críticas se realicen en el instante correcto, por un objeto o un mensaje. Lo haremos siguiendo las ideas desarrolladas en [Ölv00, ÖM02]:

```
vars T T' : Time .
var DT : DefTime .

eq delta(< I : Node | rootConDelay : DT >, T) =
    if DT == noTimeValue then < I : Node | >
    else < I : Node | rootConDelay : DT monus T > fi .
eq delta(< RAN : RandomNGen | >, T) = < RAN : RandomNGen | > .
eq delta(leader I, T) = leader I .
eq delta(from I to J be my parent with delay T, T') =
    from I to J be my parent with delay (T monus T') .
eq delta(from I to J acknowledgement with delay T, T') =
    from I to J acknowledgement with delay (T monus T') .

eq mte(< I : Node | neig : J K NEs, phase : rec >) = INF .
eq mte(< I : Node | neig : J, phase : rec >) = 0 .
eq mte(< I : Node | phase : ack >) = 0 .
eq mte(< I : Node | phase : waitParent >) = INF .
eq mte(< I : Node | phase : contention, rootConDelay : T >) = T .
eq mte(< I : Node | phase : self >) = INF .
eq mte(< RAN : RandomNGen | >) = INF .
eq mte( from I to J be my parent with delay T ) = T .
eq mte( from I to J acknowledgement with delay T ) = T .
eq mte( leader I ) = INF .
```

Por ejemplo, la primera ecuación para `delta` indica que cuando pasa un tiempo `T`, un objeto de la clase `Node` no se ve afectado si su reloj `rootConDelay` está parado, mientras que si el reloj está funcionando hay que restar a su valor el tiempo `T`. Para la operación `mte`, la primera ecuación indica que un objeto de la clase `Node` en la fase `rec` puede dejar pasar un tiempo infinito si contiene dos o más identificadores en su atributo `neig`, mientras que la segunda ecuación indica que `mte` no permite que pase el tiempo si contiene solo un identificador en dicho atributo. El resto de ecuaciones tienen una interpretación similar.

La regla tick, que permite el paso del tiempo si no hay ninguna regla que pueda ser aplicada inmediatamente, es la siguiente:

```
var C : Configuration .
crl [tick] : { C | T } => { delta(C, mte(C))  |  T plus mte(C) }
             if mte(C) =/= INF and mte(C) =/= 0 .
```

Teniendo en cuenta la definición dada de la operación `mte`, tenemos que esta regla solo puede ser aplicada cuando no hay ninguna otra disponible para ser ejecutada.

También podríamos haber modelado el paso no determinista del tiempo por medio de una regla como la siguiente:

```
rl [tick'] : { C | T } => { delta(C, T') | T plus T' } .
```

que introduce una nueva variable `T'` en el lado derecho para representar el incremento en el tiempo. Como ya sabemos, estas reglas con nuevas variables no pueden ser utilizadas directamente por el intérprete por defecto de Maude, pero sí podrían ser utilizadas al metanivel por una estrategia que instancie la variable `T'` con diferentes valores de tiempo, siempre que nos aseguremos que no dejen de ejecutarse acciones de tiempo crítico. Sin embargo, no estamos interesados en los estados intermedios entre el estado resultado de aplicación de una regla instantánea y el estado alcanzable en el que una nueva acción de tiempo crítico debe ejecutarse. De modo que si aparecen transiciones

$$s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n$$

tal que los s_j $(0 \leq j < n)$ sean estados en los que se pueda dejar pasar un tiempo t_{j+1}, y s_n es el primer estado en el que deba ocurrir una acción instantánea, modelaremos estas transiciones mediante una única transición

$$s_0 \xrightarrow{\sum_{i=1}^{n} t_i} s_n.$$

Por supuesto, esto modificará el conjunto de estados *observables*, como se comentará en la Sección 8.4.1.

8.3.3. Tercera descripción del protocolo

Hay dos consideraciones, en lo que se refiere al tiempo, que no hemos tenido en cuenta en nuestra segunda descripción. La primera tiene que ver con el hecho de que llegue a

alcanzarse un valor de tiempo definido en el estándar como `CONFIG-TIMEOUT`. Si esto ocurriera, tendríamos que la red se ha configurado incorrectamente, al contener un ciclo, por lo que debería producirse un error. La segunda consideración tiene que ver con el parámetro "raíz forzada", `fr`. En principio es posible que un nodo pase a la fase de reconocimiento `ack` cuando se han realizado $n - 1$ comunicaciones, siendo n el número de vecinos del nodo. Haciendo el atributo `fr` cierto, se fuerza a que el nodo espere en la fase `rec` *un poco más*, con la esperanza de que entre tanto se completen las n comunicaciones y el nodo se convierta en el líder.

Estas dos consideraciones afectan solo a la primera fase del protocolo, la fase de recepción de peticiones "sé mi padre".

Entonces la clase `Node`, definida en el módulo `FIREWIRE-ASYNC2`, se modifica añadiendo tres nuevos atributos:

```
class Node | neig : SetIden, children : SetIden,
             phase : Phase, rootConDelay : DefTime,
             CONFIG-TIMEOUTalarm : DefTime,
             fr : Bool, FORCE-ROOTalarm : DefTime .
```

No podemos definirla como subclase de la clase `Node` (de la sección anterior) que añada los nuevos atributos, porque en tal caso también se heredarían las reglas de la sección anterior, y algunas de ellas han de ser modificadas en la nueva descripción del protocolo.

El atributo `CONFIG-TIMEOUTalarm` es una *alarma* inicializada con el valor constante `CONFIG-TIMEOUT`, y que decrece según pasa el tiempo. Si alcanza el valor 0, el nodo se da cuenta de que la red contiene un ciclo, emitiéndose un error a través del siguiente mensaje:

```
msg error : -> Msg .
```

y el atributo `phase` se modifica a `error`, nuevo valor del tipo `Phase`.

El atributo `fr` se hace cierto cuando se pretende que el nodo sea el líder, aunque esto no garantiza que acabe siéndolo. En ese caso, el atributo `FORCE-ROOTalarm` toma como valor la constante de tiempo `FRTIME` definida en el estándar, que determina cuánto tiempo espera un nodo antes de pasar a la siguiente fase, aunque ya haya recibido peticiones de todos sus vecinos menos uno. Esta alarma también decrece cuando pasa el tiempo, y cuando alcanza el valor 0 se apaga, asignándole el valor `noTimeValue` y haciendo el atributo `fr` falso. Si el atributo `fr` es inicialmente falso, la alarma `FORCE-ROOTalarm` se inicializa a `noTimeValue`.

Veamos ahora cómo se modifican las reglas de reescritura. La regla `rec` no se modifica porque no está afectada por las nuevas consideraciones. Se añaden dos reglas que controlan lo que ocurre cuando las alarmas alcanzan el valor 0, e indican qué hacer en cada caso:

```
rl [error] :
   < I : Node | phase : rec, CONFIG-TIMEOUTalarm : 0 >
=> < I : Node | phase : error > error .
```

```
rl [stopAlarm] :
   < I : Node | phase : rec, fr : true, FORCE-ROOTalarm : 0 >
=> < I : Node | fr : false, FORCE-ROOTalarm : noTimeValue >  .
```

La regla `recN-1` sí se modifica porque ahora un nodo en la fase `rec` solo pasa a la siguiente fase si su atributo `fr` tiene el valor falso. En ese caso se apagan las dos alarmas:

```
rl [recN-1] :
   < I : Node | neig : J, children : CHs, fr : false, phase : rec >
=> < I : Node | phase : ack, CONFIG-TIMEOUTalarm : noTimeValue >  .
```

También se apagan las dos alarmas si la última petición "sé mi padre" se recibe mientras el nodo está en la fase `rec`:

```
rl [recLeader] :
   (from J to I be my parent with delay 0)
   < I : Node | neig : J, children : CHs, phase : rec >
=> < I : Node | neig : empty, children : J CHs, phase : ack, fr : false,
                FORCE-ROOTalarm : noTimeValue,
                CONFIG-TIMEOUTalarm : noTimeValue >  .
```

El resto de las reglas, que describen las siguientes fases, permanecen igual. Sin embargo, las operaciones `delta` y `mte` se han de redefinir como se indica a continuación, ya que tienen que tratar con objetos que tienen más atributos que dependen del tiempo. La interpretación intuitiva es la misma que en la sección anterior.

```
eq delta(< A : Node | phase : rec, CONFIG-TIMEOUTalarm : DT,
                      FORCE-ROOTalarm : DT' >, T) =
   if DT == noTimeValue then
     (if DT' == noTimeValue then  < A : Node | >
      else  < A : Node | FORCE-ROOTalarm : DT' monus T >
      fi)
   else
     (if DT' == noTimeValue then
            < A : Node | CONFIG-TIMEOUTalarm : DT monus T >
      else  < A : Node | CONFIG-TIMEOUTalarm : DT monus T,
                         FORCE-ROOTalarm : DT' monus T >
      fi)
   fi .
eq delta(< A : Node | phase : contention, rootConDelay : DT >, T) =
    if DT == noTimeValue then < A : Node | >
    else < A : Node | rootConDelay : DT monus T > fi .
ceq delta(< A : Node | phase : PH >, T) = < A : Node | >
    if (PH == ack or PH == waitParent or PH == self or PH == error) .
eq delta( leader I, T ) = leader I .
eq delta( error, T ) = error .
eq delta( from I to J be my parent with delay T, T') =
    from I to J be my parent with delay (T monus T') .
```

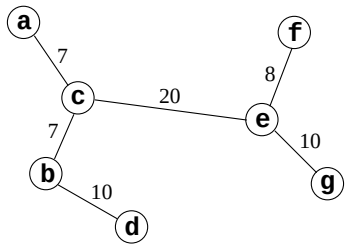


Figura 8.2: Red inicial con tiempos entre nodos.

```
eq delta( from I to J acknowledgement with delay T, T') =
    from I to J acknowledgement with delay (T monus T') .

eq mte( from I to J be my parent with delay T ) = T .
eq mte( from I to J acknowledgement with delay T ) = T .
eq mte( leader I ) = INF .
eq mte( error ) = INF .
eq mte(< A : Node | neig : I J NEs, phase : rec, CONFIG-TIMEOUTalarm : DT,
                    FORCE-ROOTalarm : DT'  >) =
   if DT == noTimeValue then
     (if DT' == noTimeValue then INF  else DT'  fi)
   else
     (if DT' == noTimeValue then DT  else  min(DT, DT')  fi)
   fi .
eq mte(< A : Node | neig : J, phase : rec, fr : true,
                    FORCE-ROOTalarm : T >) = T .
eq mte(< A : Node | neig : J, phase : rec, fr : false >) = 0 .
eq mte(< A : Node | phase : ack >) = 0 .
eq mte(< A : Node | phase : waitParent >) = INF .
eq mte(< A : Node | phase : contention, rootConDelay : T >) = T .
eq mte(< A : Node | phase : self >) = INF .
eq mte(< A : Node | phase : error >) = INF .
```

8.3.4. Ejecución de la especificación con un ejemplo

Las especificaciones del protocolo descritas en las secciones anteriores son ejecutables en el sistema Maude. Podemos sacar partido de este hecho para aumentar nuestra confianza en la corrección del protocolo. Primero definimos una configuración que denota el estado inicial de la red de la Figura 8.2, que es la misma red de la Figura 8.1(a), pero donde hemos añadido información sobre la distancia entre nodos, que refleja el tiempo que tarda un mensaje en recorrer el camino desde el emisor al receptor, utilizando la descripción del protocolo con tiempo de la Sección 8.3.2. Queremos hacer notar que las descripciones del protocolo son completamente generales, y las reglas de reescritura que las describen pueden aplicarse a cualquier red. Sin embargo, para ilustrar la ejecución del protocolo necesitamos comenzar con una red concreta, definida en una extensión del módulo con la descripción del protocolo.

```
(omod EXAMPLE is
   protecting FIREWIRE-ASYNC .

   op network7 : -> Configuration .
   op dftATTRS : -> AttributeSet .

   eq dftATTRS = children : empty, phase : rec, rootConDelay : noTimeValue .

   eq network7 = < 'Random : RandomNGen | seed : 13 >
                 < 'a : Node | neig : 'c,       dftATTRS >
                 < 'b : Node | neig : 'c 'd,    dftATTRS >
                 < 'c : Node | neig : 'a 'b 'e, dftATTRS >
                 < 'd : Node | neig : 'b,       dftATTRS >
                 < 'e : Node | neig : 'c 'f 'g, dftATTRS >
                 < 'f : Node | neig : 'e,       dftATTRS >
                 < 'g : Node | neig : 'e,       dftATTRS > .

   eq timeLink('a,'c) = 7  .  eq timeLink('c,'a) = 7  .
   eq timeLink('b,'c) = 7  .  eq timeLink('c,'b) = 7  .
   eq timeLink('b,'d) = 10 .  eq timeLink('d,'b) = 10 .
   eq timeLink('c,'e) = 20 .  eq timeLink('e,'c) = 20 .
   eq timeLink('e,'f) = 8  .  eq timeLink('f,'e) = 8  .
   eq timeLink('e,'g) = 10 .  eq timeLink('g,'e) = 10 .
endom)
```

Ahora podemos pedir al sistema Maude que reescriba la configuración inicial utilizando su estrategia por defecto:

```
Maude> (rew { network7 | 0 } .)

result ClockedSystem : { leader 'c
  < 'e : Node | neig : 'c, restATTRS > < 'd : Node | neig : 'b, restATTRS >
  < 'a : Node | neig : 'c, restATTRS > < 'f : Node | neig : 'e, restATTRS >
  < 'b : Node | neig : 'c, restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'c : Node | neig : empty, restATTRS >
  < 'Random : RandomNGen | seed : 9655 >  | 920 }
```

donde para hacer más legible la presentación del término resultante hemos representado por medio de `restATTRS` los atributos que son iguales para todos los nodos, de modo que tenemos

```
restATTRS = children : empty, phase : self, rootConDelay : noTimeValue
```

Obsérvese que en la configuración alcanzada aparece un único mensaje `leader` que declara al nodo `c` como líder.

8.4. Análisis exhaustivo de estados

Las propiedades deseables que debe cumplir el protocolo son dos: un único líder es elegido (seguridad) y eventualmente se elige algún líder (vivacidad). La ejecución proporcionada por el intérprete por defecto de Maude, al seguir solo uno de los caminos posibles, no es suficiente para asegurar que estas propiedades se cumplen siempre, en cualquier ejecución posible. Podemos ir más allá en nuestro análisis realizando una exploración exhaustiva de todos los posibles comportamientos del protocolo.

En esta sección mostramos cómo pueden utilizarse las características reflexivas de Maude, de forma similar a como se utilizaron en el Capítulo 4 para hacer ejecutable la semántica de CCS, para demostrar que las especificaciones del protocolo funcionan de la forma esperada cuando son aplicadas a una red inicial concreta arbitraria. Esto se hace comprobando que las dos propiedades se cumplen al final del protocolo en *todo* posible comportamiento de este, partiendo de la configuración inicial que representa la red en cuestión.

8.4.1. Estrategia de búsqueda

Validaremos nuestras especificaciones mediante una exploración exhaustiva de todos los posibles comportamientos en el árbol de posibles reescrituras del término que representa el estado inicial de una red. Se buscan todos los términos alcanzables irreducibles y se comprueba que en todos ellos solo se ha elegido un líder. La estrategia de búsqueda está basada en los trabajos presentados en [BMM98, CDE+00c], y es muy similar a la utilizada en la Sección 4.3.2 para hacer ejecutable la semántica de CCS. La estrategia es completamente general en el sentido de que no depende del módulo de sistema concreto cuyas reglas se utilizan para construir el árbol de búsqueda.

El módulo que implementa la estrategia de búsqueda está parametrizado con respecto a una constante que será la metarrepresentación del módulo Maude con el que queramos trabajar. Así pues, definimos una teoría parámetro con una constante `MOD` que represente este módulo, y una constante `labels` que represente la lista de etiquetas de reglas de reescritura que pueden aplicarse:

```
(fth AMODULE is
  including META-LEVEL .

  op MOD : -> Module .
  op labels : -> QidList .
endfth)
```

El módulo que contiene la estrategia, y que extiende al módulo `META-LEVEL`, es entonces el módulo parametrizado `SEARCH[M :: AMODULE]`. La estrategia controla las posibles reescrituras de un término por medio de la función del metanivel `meta-apply`. Esta devuelve *una* de las posibles reescrituras en un paso al nivel más alto del término dado. Comenzamos definiendo una operación `allRew` que devuelve *todas* las posibles reescritu-

ras *secuenciales en un paso* [Mes92] de un término dado `T`, utilizando reglas de reescritura con etiquetas en la lista `labels`.

Las operaciones necesarias para encontrar todas las posibles reescrituras son las siguientes[2]:

```
op allRew : Term QidList -> TermList .
op topRew : Term Qid MachineInt -> TermList .

var T : Term .
var L : Qid .
var LS : QidList .

eq allRew(T, nil) = ~ .
eq allRew(T, L LS) = topRew(T, L, 0) , allRew(T, LS) .

eq topRew(T, L, N) =
    if extTerm(meta-apply(MOD, T, L, none, N)) == error* then ~
    else (extTerm(meta-apply(MOD, T, L, none, N)) ,
          topRew(T, L, N + 1))
    fi .
```

Ahora podemos definir una operación `allSol` que busque, en el árbol formado por todas las posibles reescrituras de un término `T`, los términos irreducibles, que no son otra cosa que las hojas del árbol de búsqueda.

```
sort TermSet .
subsort Term < TermSet .
op `{`} : -> TermSet .
op _U_ : TermSet TermSet -> TermSet [assoc comm id: {}] .
eq T U T = T .

op allSol : Term -> TermSet .
op allSolDepth : TermList -> TermSet .

var TL : TermList .

eq allSol(T) = allSolDepth(meta-reduce(MOD,T)) .
eq allSolDepth(~) = {} .
eq allSolDepth( T ) =
    if allRew(T, labels) == ~ then  T
    else  allSolDepth(allRew(T, labels)) fi .
eq allSolDepth( (T, TL) ) =
    if allRew(T, labels) == ~ then ( T U allSolDepth(TL) )
    else  allSolDepth((allRew(T, labels), TL)) fi .
```

[2]Obsérvese que solo se reescribe al nivel más alto de un término, nunca sus argumentos. Esto es suficiente en las aplicaciones de la estrategia a las descripciones del protocolo, donde se reescriben los estados completos de la red, de tipo `ClockedSystem`.

Antes de ver un ejemplo, vamos a considerar dos posibles modificaciones de esta estrategia. Supondremos primero que hemos separado las reglas que definen el protocolo en reglas impacientes y reglas perezosas, como se comentó en la Sección 8.3.1, y que tenemos constantes `lazyLabels` y `eagerLabels` identificando las listas de etiquetas de reglas en cada uno de estos grupos. Entonces podemos modificar la operación `allSolDepth` para asegurar que las reglas impacientes se aplican primero, y que las reglas perezosas solo se aplican cuando ya no se puede aplicar ninguna regla impaciente.

```
eq allSolDepth( T ) =
    if allRew(T, eagerLabels) == ~ then
      (if allRew(T, lazyLabels) == ~ then T
       else allSolDepth(allRew(T, lazyLabels)) fi)
    else  allSolDepth(allRew(T, eagerLabels)) fi .
eq allSolDepth( (T, TL) ) =
    if allRew(T, eagerLabels) == ~ then
      (if allRew(T, lazyLabels) == ~ then ( T U allSolDepth(TL) )
       else allSolDepth((allRew(T, lazyLabels), TL)) fi)
    else  allSolDepth((allRew(T, eagerLabels), TL)) fi .
```

En segundo lugar, la estrategia también puede modificarse para mantener, para cada término `T`, los pasos de reescritura que se han realizado para alcanzar `T` desde el término inicial. Esto se hace en [DMT98] manteniendo, en vez de términos de tipo `Term`, términos de tipo `Path` construidos con la siguiente sintaxis:

```
sorts Path Step .
subsort Term < Path .

op path : Path Step -> Path .
op step : Qid Term ->  Step .
```

De esta forma, el término `path(path(`t_0`, step(`l_1`,`t_1`)), step(`l_2`,`t_2`))` indica que el término t_2 ha sido alcanzado desde t_0 de la siguiente manera $t_0 \xrightarrow{l_1} t_1 \xrightarrow{l_2} t_2$.

Esto es útil en el caso de que se encuentre un error cuando se valida el protocolo; en tal caso, el camino que conduce a la configuración errónea muestra *un contraejemplo* de la corrección del protocolo (véase [DMT98]).

En la aplicación concreta de esta estrategia de búsqueda utilizada en la siguiente sección, los términos en los nodos del árbol son de tipo `ClockedSystem`, y los hijos de un nodo X son los términos alcanzables desde X por aplicación de alguna de las reglas de reescritura en el módulo `EXAMPLE`. Debido a nuestro tratamiento del tiempo (véase la Sección 8.3), si un nodo X del árbol de búsqueda tiene más de un hijo es porque X representa una configuración donde pueden realizarse varias acciones de tiempo crítico de forma no determinista. Por el contrario, si X representa una configuración donde puede pasar el tiempo, solo tendrá un hijo X', alcanzable a partir de X por aplicación de la regla de reescritura `tick` (Sección 8.3.2) y donde X' representará una configuración sobre la que tiene que realizarse una acción de tiempo crítico.

El no determinismo y los múltiples encajes de patrones módulo asociatividad son las razones de la explosión de estados en el árbol de búsqueda. Esto, y el hecho de que estemos moviéndonos continuamente entre el nivel objeto y el metanivel a la hora de ir generando y recorriendo el árbol de búsqueda, hace que el análisis de estados resulte un tanto ineficiente. Esta ineficiencia desaparece cuando utilizamos las nuevas facilidades de Maude 2.0, como veremos en la Sección 8.7.

8.4.2. Utilización de la estrategia con un ejemplo

Ahora veremos cómo se puede utilizar la estrategia para demostrar que la descripción con tiempo del protocolo siempre funciona bien cuando se aplica a la red concreta del módulo `EXAMPLE` (Sección 8.3.4). Para instanciar el módulo genérico `SEARCH`, necesitamos la metarrepresentación del módulo `EXAMPLE`. Utilizamos la función `up` de Full Maude para obtener la metarrepresentación de un módulo o un término [CDE+99].

```
(mod META-FIREWIRE is
  including META-LEVEL .
  op METAFW : -> Module .
  eq METAFW = up(EXAMPLE) .
endm)
```

y declaramos una vista e instanciamos el módulo genérico `SEARCH` con ella:

```
(view ModuleFW from AMODULE to META-FIREWIRE is
  op MOD to METAFW .
  op labels to ('rec 'recN-1 'recLeader 'ack 'ackLeader
                'ackParent 'wait1 'wait2 'contenReceive
                'contenSend 'tick) .
endv)

(mod SEARCH-FW is
  protecting SEARCH[ModuleFW] .
endm)
```

Ahora podemos probar con el ejemplo. Utilizamos el comando `down` para presentar la salida de forma más legible. El resultado en Maude es el siguiente:

```
Maude> (down EXAMPLE : red allSol(up(EXAMPLE, { network7 | 0 })) .)

rewrites: 155131 in 50343ms cpu (50707ms real) (3081 rewrites/second)

result ClockedSystem : { leader 'c
  < 'e : Node | neig : 'c, restATTRS > < 'd : Node | neig : 'b, restATTRS >
  < 'a : Node | neig : 'c, restATTRS > < 'f : Node | neig : 'e, restATTRS >
  < 'b : Node | neig : 'c, restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'c : Node | neig : empty, restATTRS >
  < 'Random : RandomNGen | seed : 9655 >  | 920 }
```

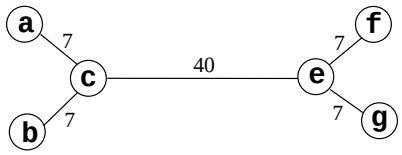


Figura 8.3: Una red simétrica.

Podemos observar que solo se ha elegido un líder y que la configuración alcanzada es la de la Figura 8.1(d). No es sorprendente que solo se alcance una configuración (módulo idempotencia) aunque se comprueben todos los posibles comportamientos, ya que el resultado del protocolo no es tan indeterminista cuando se le añade tiempo. Las restricciones temporales suponen restricciones severas sobre las configuraciones finales alcanzables.

Ahora probamos el protocolo sobre una red a partir de la cual se alcanzan dos configuraciones finales. La red será la mostrada en la Figura 8.3 y el módulo Maude que la define es el siguiente:

```
(omod EXAMPLE2 is
    protecting FIREWIRE-ASYNC .

    op network6 : -> Configuration .
    op dftATTRS : -> AttributeSet .

    eq dftATTRS = children : empty, phase : rec,
                  rootConDelay : noTimeValue .

    eq network6 = < 'a : Node | neig : 'c,       dftATTRS >
                  < 'b : Node | neig : 'c,       dftATTRS >
                  < 'c : Node | neig : 'a 'b 'e, dftATTRS >
                  < 'e : Node | neig : 'c 'f 'g, dftATTRS >
                  < 'f : Node | neig : 'e,       dftATTRS >
                  < 'g : Node | neig : 'e,       dftATTRS >
                  < 'Random : RandomNGen | seed : 13 > .

    eq timeLink('a,'c) =  7 .
    eq timeLink('b,'c) =  7 .
    eq timeLink('c,'e) = 40 .
    eq timeLink('e,'f) =  7 .
    eq timeLink('e,'g) =  7 .

    var I J : Qid .
    ceq timeLink(J,I) = timeLink(I,J) if I < J .
endom)
```

Después de instanciar la estrategia de búsqueda con la metarrepresentación de este nuevo módulo `EXAMPLE2`, podemos pedir a Maude que busque todas las configuraciones finales alcanzables. Como se esperaba en todas ellas se ha elegido un único líder.

```
Maude> (down EXAMPLE : red allSol(up(EXAMPLE, { network6 | 0 })) .)

rewrites: 288781 in 100791ms cpu (102636ms real) (2865 rewrites/second)

result ClockedSystem : { leader 'c
  < 'e : Node | neig : 'c,    restATTRS > < 'a : Node | neig : 'c, restATTRS >
  < 'c : Node | neig : empty, restATTRS > < 'b : Node | neig : 'c, restATTRS >
  < 'f : Node | neig : 'e,    restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'Random : RandomNGen | seed : 9655 > | 997 }
& { leader 'e
  < 'e : Node | neig : empty, restATTRS > < 'a : Node | neig : 'c, restATTRS >
  < 'c : Node | neig : 'e,    restATTRS > < 'b : Node | neig : 'c, restATTRS >
  < 'f : Node | neig : 'e,    restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'Random : RandomNGen | seed : 9655 > | 997 }
```

La presencia de un único mensaje de líder en las configuraciones finales no excluye la posibilidad de que se hubieran declarado, y dejado de declarar, una secuencia de líderes. Pero un simple examen de las reglas de reescritura muestra que ningún mensaje de líder aparece en la parte izquierda de ninguna regla de reescritura, y no hay ninguna operación que elimine mensajes de esta clase de una configuración. Por tanto, concluimos que una vez que ha aparecido un mensaje de líder, este nunca desaparece.

La estrategia de análisis presentada tiene que ejecutarse comenzando con una red concreta, aunque esta pueda ser elegida arbitrariamente. Por tanto, deberíamos ejecutar la estrategia con muchas redes diferentes para estar seguros de que el protocolo funciona correctamente en todos los casos. Esta es la razón por la cual en [VPMO02] hemos propuesto una tercera clase de análisis que evite la necesidad de comenzar con una red concreta. Este trabajo es parte de la tesis doctoral de Isabel Pita [Pit03]. En la próxima sección resumimos las principales ideas.

8.5. Demostración formal por inducción

Las demostraciones por inducción presentadas en [VPMO02] prueban que al ejecutar cualquiera de las especificaciones del protocolo descritas en las secciones anteriores sobre una red conexa y acíclica cualquiera, se selecciona un líder y solo uno.

La idea principal de estas demostraciones formales por inducción [Pit03] consiste en definir *observaciones* que permitan expresar propiedades de una configuración del sistema. Después se observan los cambios hechos por las reglas de reescritura en las configuraciones, probando que ciertas propiedades son invariantes.

Para la primera descripción simple con comunicación síncrona (Sección 8.2) se define una única observación *nodes*, el cual es un conjunto de pares $\langle A; S\rangle$ donde A es un identificador de nodo de la red y S es el conjunto de nodos con los que todavía A no se ha comunicado. A partir de propiedades básicas que satisface el conjunto *nodes*, se pueden derivar por inducción las propiedades deseables del protocolo.

Para la segunda y tercera descripciones (Secciones 8.3.2 y 8.3.3) las observaciones se complican, ya que los objetos en las descripciones tienen muchos más atributos, pasan por

diferentes fases, y hay muchas más reglas de reescritura. Básicamente, se tienen observaciones diferentes para los conjuntos de nodos en las diferentes fases. Se puede probar que si los conjuntos no son vacíos, se puede aplicar alguna regla, y que una medida natural que depende del número de elementos en los conjuntos decrece. Esto prueba la vivacidad y terminación del protocolo. También se prueban propiedades de justicia, como que solo se puede seleccionar un líder por medio de las reglas de reescritura. Para la tercera descripción se prueba además que si existe un ciclo en la red, se genera un mensaje de error.

Una descripción detallada de estas demostraciones puede encontrarse en [VPMO02, Pit03].

8.6. Evaluación

En la convocatoria para enviar artículos para el volumen especial de la revista *Formal Aspects of Computing* sobre la aplicación de métodos formales al estándar IEEE 1394, se pedía a los autores que evaluaran sus contribuciones considerando una serie de cuestiones [MRS02]. He aquí el resultado de nuestra evaluación.

Hemos especificado todas las partes de este protocolo de una manera formal, utilizando la lógica de reescritura tal y como la implementa el sistema Maude. Consideramos que el enfoque seguido por nosotros, orientado a objetos con tiempo, es particularmente expresivo para esta clase de protocolos. También hemos analizado el protocolo completo de dos formas diferentes. La primera es completamente automática y consiste en la exploración exhaustiva de todos los estados alcanzables a partir del estado inicial de una red concreta (pero elegida arbitariamente). La otra es manual y proporciona una demostración matemática de los requisitos de seguridad y vivacidad para cualquier red.

También creemos que nuestra solución es fácil de modificar o extender. De hecho, hemos presentado tres especificaciones a diferentes niveles de abstracción. Por el contrario, las demostraciones formales son bastante sensibles a los cambios, aunque también son modulares, lo que ha permitido reutilizar y extender la demostración de la segunda descripción cuando se verificaba la tercera.

Las especificaciones fueron realizadas por el autor de esta tesis en un plazo de dos semanas, cuando ya tenía gran experiencia en el uso de Maude, pero con menor conocimiento sobre la introducción de aspectos de tiempo en la lógica de reescritura, el cual se adquirió durante dicho periodo. Las demostraciones formales requirieron tres semanas más.

Todos los conceptos involucrados en este marco de especificación son fáciles de entender por un ingeniero medio, por lo que pensamos que cualquiera con mínimos conocimientos de reescritura y especificaciones algebraicas podría comprender rápidamente nuestras soluciones.

8.7. Ejecución de las especificaciones en Maude 2.0

Cuando realizamos la especificación y verificación de este protocolo en Maude no teníamos disponibles las facilidades que ofrece la nueva versión Maude 2.0. Por eso tuvimos que implementar al metanivel la estrategia de búsqueda presentada en la Sección 8.4.1, que ejecuta la especificación del protocolo de todas las formas posibles y devuelve todos los estados irreducibles alcanzables.

En Maude 2.0 podemos realizar una búsqueda en el árbol de posibles reescrituras de un término mediante el comando `search` al nivel objeto (y la operación `metaSearch` al metanivel), como hemos hecho en el Capítulo 5 para ejecutar la representación de la semántica de CCS. El comando

$$\texttt{search}\ T\ \texttt{=>!}\ P\ \texttt{.}$$

busca en el árbol de posibles reescrituras del término T aquellos términos irreducibles que encajan con el patrón P. Esto es justo lo que necesitamos en nuestro análisis exhaustivo de estados. Dada una configuración inicial de la red, tenemos que buscar todos los estados irreducibles alcanzables, para comprobar después que en todos ellos se ha declarado un único líder.

Habiendo adaptado los módulos con las especificaciones del protocolo a la nueva versión de Maude, podemos ejecutar los ejemplos vistos en la Sección 8.4.2.

El primer ejemplo busca todos los estados irreducibles alcanzables a partir de la configuración inicial de la red de la Figura 8.2. La única configuración alcanzable es la misma que obtuvimos en la Sección 8.4.2.

```
Maude> (search { network7 | 0 } =>! CS:ClockedSystem .)

rewrites: 18339 in 0ms cpu (158ms real) (~ rewrites/second)

Solution 1
CS:ClockedSystem <- { leader 'c
  < 'e : Node | neig : 'c, restATTRS > < 'd : Node | neig : 'b,    restATTRS >
  < 'g : Node | neig : 'e, restATTRS > < 'f : Node | neig : 'e,    restATTRS >
  < 'a : Node | neig : 'c, restATTRS > < 'c : Node | neig : empty, restATTRS >
  < 'b : Node | neig : 'c, restATTRS >
  < 'Random : RandomNGen | seed : 9655 >  | 920 }

No more solutions.
```

Realizar la ejecución análoga con la estrategia de búsqueda definida al metanivel requería más de 50 segundos, mientras que Maude 2.0 necesita solamente 158 milisegundos.

El segundo ejemplo utiliza la red de la Figura 8.3. Las dos configuraciones alcanzables son las mismas que obtuvimos en la Sección 8.4.2. En esta ejecución pasamos de 102 segundos a 167 milisegundos. Como se observa, la ganancia obtenida al tener el comando de búsqueda implementado directamente en el sistema es más que notable, como era de esperar.

```
Maude> (search { network6 | 0 } =>! CS:ClockedSystem .)

rewrites: 20637 in 0ms cpu (167ms real) (~ rewrites/second)

Solution 1
CS:ClockedSystem <- { leader 'c
  < 'e : Node | neig : 'c,    restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'f : Node | neig : 'e,    restATTRS > < 'a : Node | neig : 'c, restATTRS >
  < 'c : Node | neig : empty, restATTRS > < 'b : Node | neig : 'c, restATTRS >
  < 'Random : RandomNGen | seed : 9655 > | 997 }

Solution 2
CS:ClockedSystem <- { leader 'e
  < 'e : Node | neig : empty, restATTRS > < 'g : Node | neig : 'e, restATTRS >
  < 'f : Node | neig : 'e,    restATTRS > < 'a : Node | neig : 'c, restATTRS >
  < 'c : Node | neig : 'e,    restATTRS > < 'b : Node | neig : 'c, restATTRS >
  < 'Random : RandomNGen | seed : 9655 > | 997 }

No more solutions.
```

8.8. Comparación con descripciones en E-LOTOS

En [SV01] utilizamos el álgebra de procesos E-LOTOS para indicar cómo describir a distintos niveles de abstracción un protocolo de comunicaciones, presentando como caso de estudio el protocolo de elección de líder considerado en este capítulo. El énfasis principal se puso en la especificación, ilustrando los operadores que proporciona E-LOTOS, particularmente aquellos relacionados con el paralelismo y el tiempo, y el rango de niveles de abstracción disponibles.

E-LOTOS es una técnica de descripción formal y estándar de ISO [ISO01], desarrollada a partir del estándar LOTOS [ISO89] que, como hemos visto en el Capítulo 7, permite la especificación de comportamiento y de datos utilizando un álgebra de procesos y tipos abstractos de datos. E-LOTOS mantiene la fuerte base formal de su predecesor LOTOS, que fue desarrollado para la descripción y análisis de sistemas distribuidos y protocolos de comunicación. E-LOTOS mantiene este enfoque, añadiendo mayor expresividad y poder de estructuración, y siendo más fácil de utilizar por gente no experta. Las mejoras más importantes que se incorporan son las siguientes:

- *La noción de tiempo.* En E-LOTOS se puede definir el tiempo exacto en el que ocurren los eventos.
- *Un nuevo lenguaje de datos.* Se proporciona una descripción de tipos de datos similar a la utilizada en lenguajes de programación funcionales, manteniendo el soporte formal. Los datos se manipulan mediante funciones.
- *Modularidad.* Se permite la definición de tipos, funciones y procesos en módulos separados, el control de su visibilidad por medio de interfaces y la definición de módulos genéricos.

- *Nuevos operadores que mejoran su poder expresivo.* Concretamente, se han añadido operadores para tratar las excepciones y otros para representar esquemas de paralelismo complejos de una forma sencilla.

- *Construcciones de lenguajes de programación imperativos.* Estas hacen que el lenguaje sea útil para cubrir las últimas fases del ciclo de desarrollo del software, cuando se desarrollan las implementaciones, y también hacen más fácil la especificación de sistemas a gente acostumbrada a utilizar tales lenguajes.

- *Variables mutables.* E-LOTOS tiene variables a las que se puede asignar distintos valores, como en un lenguaje de programación imperativo.

Un tratamiento más detallado de E-LOTOS y sus características puede encontrarse en nuestros trabajos [HLDRV99, HLDQ+01], donde se explican las principales implicaciones de la inclusión de estos elementos y se comentan algunas de las alternativas presentadas durante la fase de estandarización del lenguaje.

Como hemos dicho, en [SV01] utilizamos E-LOTOS para describir el protocolo de elección de líder del estándar IEEE 1394 a diferentes niveles de abstracción. Las entidades principales de E-LOTOS son los tipos de datos, los procesos y los eventos (comunicaciones entre procesos). En el protocolo, los elementos principales son los nodos y las comunicaciones entre ellos, que se modelan de forma natural como procesos y eventos, respectivamente. Los tipos de datos se utilizan para modelar el estado interno de los nodos y los valores comunicados. Por ejemplo, los diferentes mensajes que aparecen en el protocolo se representan como diferentes valores del tipo que se utiliza para los canales de comunicación entre procesos.

En [SV01] se presentan cuatro especificaciones. La primera es la más abstracta, limitándose a indicar que se ha de producir un evento `leader`, es decir, que se ha seleccionado un líder. En Maude no hemos dado ninguna especificación tan abstracta. El resto de especificaciones corresponden a las presentadas anteriormente en las Secciones 8.2, 8.3.2 y 8.3.3.

Así pues, la segunda descripción es una descripción síncrona sin tiempo. Aunque hay que decir que en E-LOTOS la noción de tiempo está siempre presente, aunque no se utilicen explícitamente los operadores temporales. En el lenguaje existe también la noción de evento *urgente* (o instantáneo, como hemos descrito en la Sección 8.3.1), por lo que se logra la evolución del sistema haciendo que algunos eventos sean urgentes. El problema con esta segunda descripción en E-LOTOS es que depende de la red concreta que se quiera probar; en particular, del número de nodos que la misma tenga. Esto es debido a que en E-LOTOS se especifican de forma mucho más concreta las comunicaciones entre procesos y qué procesos pueden comunicarse entre sí. En Maude, ninguna de las especificaciones dependía de una red concreta, ni del número de nodos de esta.

La tercera descripción presentada en E-LOTOS es una versión con comunicación asíncrona en la que se tiene que tratar el problema del conflicto de raíz. Los nodos van pasando por diferentes fases modeladas por diferentes procesos mutuamente recursivos. También se representan explícitamente los canales de comunicación (de un único sentido)

entre cada par de procesos. Esta descripción, gracias a los nuevos operadores de paralelismo de E-LOTOS, no depende de la red concreta ni del número de nodos de la misma. El número de nodos y las conexiones concretas entre ellos son parámetros de la especificación.

La cuarta y última descripción presentada en E-LOTOS modifica la fase de recepción de peticiones para que un nodo sea capaz de detectar ciclos en la red y para tratar el parámetro `fr`. Al haberse especificado en la descripción anterior cada fase mediante un proceso diferente, solo uno de ellos tiene que ser modificado en esta extensión.

Debido a la falta de herramientas que soportaran el lenguaje E-LOTOS, al hacer nuestro trabajo con este lenguaje no pudimos realizar ningún tipo de análisis sobre las especificaciones (aparte de la ejecución manual de los procesos utilizando la definición formal de la semántica operacional). Desgraciadamente, esta carencia se mantiene en la actualidad. Como comentaremos en el Capítulo 10, esta es una de las tareas en las que nos gustaría participar en un futuro próximo.

Al comparar E-LOTOS con otros formalismos utilizados para describir el protocolo, como los autómatas I/O [DGRV97] y el álgebra de procesos μCRL [SZ98], las conclusiones fueron que E-LOTOS es más útil para escribir especificaciones que tenga que leer gente menos experta en formalismos, debido al alto contenido matemático de los otros formalismos.

Las descripciones dadas en Maude, utilizando especificaciones orientadas a objetos, estarían entre ambos extremos. Sin llegar al detalle de las especificaciones E-LOTOS, mantienen una interpretación intuitiva, clara y sencilla de las reglas de reescritura. Además se pueden ejecutar directamente y se pueden analizar formalmente.

8.9. Conclusiones

A las conclusiones ya comentadas en las Secciones 8.6 y 8.8 nos gustaría añadir las siguientes.

En este capítulo hemos mostrado cómo se puede utilizar Maude, y en particular sus especificaciones orientadas a objetos, como un lenguaje de especificación formal para describir y analizar a diferentes niveles de abstracción un protocolo de comunicación como el protocolo de elección de líder del "FireWire". Hemos puesto especial énfasis en el tratamiento de los aspectos relacionados con el tiempo, esenciales en este protocolo, ya que como comentamos en la Sección 8.3, si así no lo hiciéramos llegaríamos a tener una descripción incorrecta desde el punto de vista de la ejecución.

También hemos visto cómo enfocar desde dos puntos de vista diferentes la demostración de la corrección del protocolo respecto a las propiedades deseables de seguridad y vivacidad. Uno de ellos, descrito en la Sección 8.4, consiste en la exploración exhaustiva de todos los estados irreducibles alcanzables aplicando de todas las formas posibles las reglas del protocolo a una configuración inicial de una red conexa y acíclica. Utilizando técnicas similares a las utilizadas en el Capítulo 4 para hacer ejecutable la semántica de CCS, hemos implementado una estrategia de búsqueda que realiza esta exploración, comprobando que el protocolo se comporta correctamente en cualquiera de sus múltiples ejecuciones.

El segundo método de demostración prueba por inducción que la ejecución del protocolo sobre cualquier red conexa y acíclica conduce a la elección de un único líder. Este método se ha resumido en la Sección 8.5, y forma parte de la tesis doctoral de Isabel Pita [Pit03].

Capítulo 9

Representación de RDF en Maude

Recientemente se ha incrementado de forma notable el interés en la web semántica [BLHL01] y en asuntos relacionados con la especificación y explotación de la semántica en la *World Wide Web*. De hecho ha habido importantes avances para tratar la semántica formal de la información de manera que se facilite su procesamiento por parte de las máquinas. En esta categoría podemos incluir todo el trabajo que se está realizando por parte del grupo de RDF del W3C [W3C].

La web actual, concebida desde el punto de vista de los humanos, está aún codificada en gran parte en HTML. En los últimos años, se ha propuesto XML [XML] como una codificación alternativa enfocada principalmente al procesamiento por parte de las máquinas, convirtiéndose en el estándar para el intercambio de información en Internet. Sin embargo, esta no es la solución final, ya que solo proporciona soporte para la representación sintáctica de la información, y no de su significado. Esencialmente proporciona un mecanismo para declarar y utilizar estructuras de datos simples y por tanto no es un lenguaje apropiado para expresar conocimiento más complejo. Mejoras recientes de XML básico, tales como XML Schema [XML01], no resultan suficientes para cubrir las necesidades del lenguaje que necesita la web semántica.

RDF (marco de descripción de recursos, en inglés *Resource Description Framework*) [LS99] y RDFS (esquemas RDF, en inglés *RDF Schema*) [BG02] representan un intento de resolución de estas deficiencias construido por encima de XML, aunque siguen siendo bastante limitados como lenguaje de representación de conocimiento. En particular, la semántica sigue estando pobremente especificada. De cualquier forma, representan el primer paso para construir la web semántica, permitiendo que todas las partes implicadas tengan una comprensión común de los datos.

Tim Berners-Lee concibe la web semántica que integra estos conceptos como la arquitectura por capas [BLHL01] que se muestra en la Figura 9.1. En el nivel más bajo RDF proporciona un modelo de datos simples y una sintaxis estandarizada para *metadatos* (datos sobre los datos) sobre los recursos web. El siguiente nivel es la capa de los *esquemas* con la definición de vocabularios. Los siguientes niveles se acercan a la capa lógica proporcionada por un lenguaje de representación del conocimiento. Lo más importante es que cada capa es una extensión de RDF.

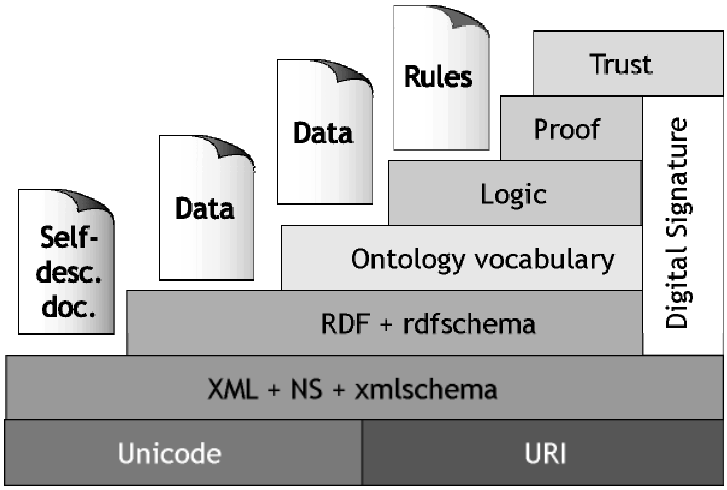


Figura 9.1: Arquitectura por niveles de la web semántica según Tim Berners-Lee.

Nuestro objetivo a largo plazo consiste en tener una forma correcta de verificar y validar las interacciones que pueden llevar a cabo aplicaciones y agentes en un entorno distribuido dentro de la web semántica. Para conseguirlo, un primer paso importante consiste en dar semántica a RDF y RDFS. Basándose en este soporte formal se podrán analizar propiedades, así como realizar transformaciones y verificaciones.

En este capítulo damos un soporte semántico a RDF por medio del lenguaje formal Maude. En concreto, utilizamos Maude para:

- dar semántica a los documentos RDF por medio de una traducción a modulos orientados a objetos en Maude;
- implementar esta traducción; e
- implementar las aplicaciones que hacen uso de los documentos traducidos.

Los documentos RDF pueden traducirse fácilmente a módulos orientados a objetos en Maude y por tanto pueden convertirse en datos para aplicaciones Maude, como mostraremos en las Secciones 9.2 y 9.5. Esta representación proporciona una semántica formal para los documentos RDF (la de los módulos traducidos) y permite que los programas que manipulen estos documentos puedan ser expresados en el mismo formalismo, al ser las especificaciones Maude directamente ejecutables.

La traducción de los documentos RDF a Maude puede implementarse también en el propio Maude, por medio de una función que recibe un documento RDF y devuelve un módulo orientado a objetos, de la misma forma que en el Capítulo 7 vimos una traducción de ACT ONE a Maude.

Por último, los documentos RDF traducidos pueden integrarse en una aplicación basada en agentes web escrita utilizando Mobile Maude, una extensión de Maude para espe-

cificar sistemas basados en agentes móviles. Es decir, se utiliza el mismo marco semántico para traducir documentos RDF y para expresar los programas que los manipulan.

Una de las características más importantes de nuestro enfoque es que es completamente orientado a objetos. Cierto es que, como veremos en la Sección 9.2, se utiliza una idea algo relajada de objeto, al igual que en RDF, donde no todos los atributos de una clase tienen que ser definidos para los objetos que pertenecen a esa clase, pero toda la potencia de la orientación a objetos está soportada, incluyendo la herencia. Como veremos en la Sección 9.5, esto es crucial para que descripciones RDF que sean más específicas (esto es, con más atributos definidos) puedan ser utilizadas por agentes que requirien menos información que la proporcionada. Esto da soporte a la visión de la web semántica por la cual *datos semánticamente ricos pueden ser vistos parcialmente por agentes semánticamente más pobres.*

Una versión preliminar de parte del trabajo descrito en este capítulo se presentó en la *First International Semantic Web Conference, ISWC 2002* [BLMO+02a], en forma de póster, y una versión más completa se ha presentado en las *Segundas Jornadas sobre Programación y Lenguajes, PROLE 2002*, dando lugar a la publicación [BLMO+02b].

9.1. RDF y RDFS

El marco de descripción de recursos RDF (*Resource Description Framework*) es un lenguaje de propósito general para representar información en la *World Wide Web*. Proporciona un marco común en el que expresar la misma de forma que pueda ser intercambiada entre aplicaciones sin pérdida de conocimiento, proporcionando una forma sencilla de expresar propiedades de *recursos web*, es decir, de objetos identificables de forma única por medio de un URI (identificador de recursos uniforme, en inglés *Uniform Resource Identifier* [URI00]).

RDF se basa en la idea de que los elementos que queremos describir tienen propiedades que a su vez tienen valores (los cuales pueden ser literales u otros recursos), y que los recursos pueden ser descritos mediante sentencias que especifican estas propiedades y sus valores. Una sentencia tiene tres elementos: un recurso específico (sujeto), una propiedad (predicado), y el valor de dicha propiedad para ese recurso (objeto); por ejemplo, en la siguiente sentencia

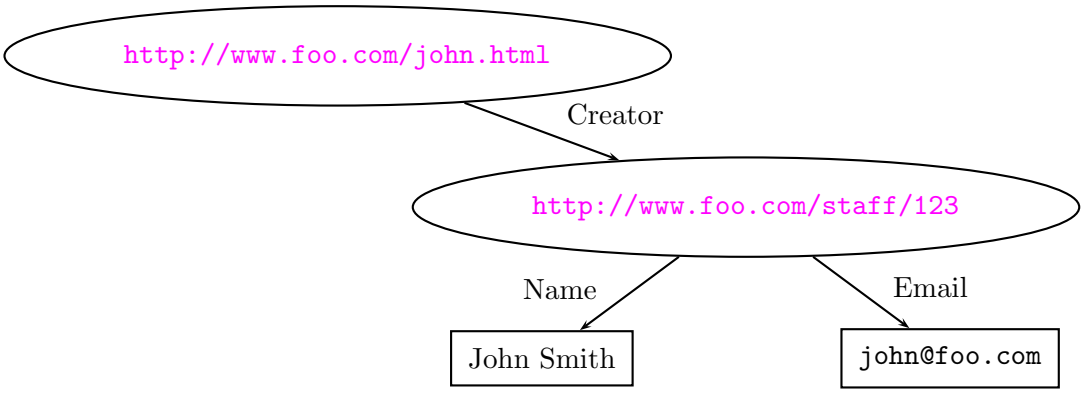
> "El creador de la página web `http://www.foo.com/john.html` es John Smith."

el sujeto es `http://www.foo.com/john.html`, la propiedad es "creador", y el valor de dicha propiedad es "John Smith".

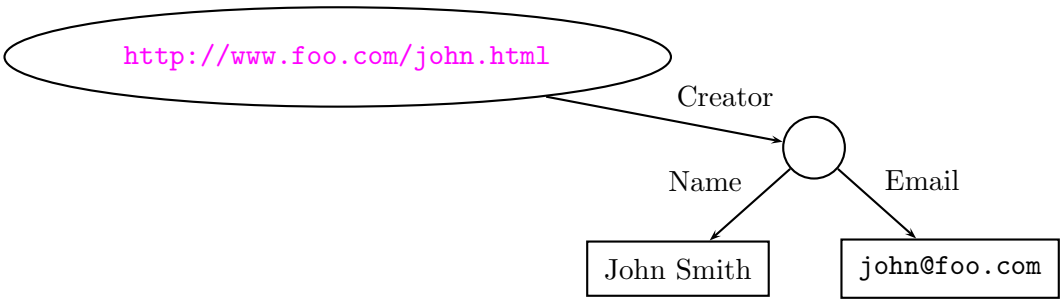
Una colección de tales sentencias referidas a un mismo recurso se denomina *descripción*. En [LS99] se presenta un modelo independiente de la sintaxis para representar recursos y sus correspondientes descripciones. Este modelo representa las relaciones entre recursos, sus propiedades y sus valores en un grafo etiquetado dirigido: los recursos se identifican mediante nodos elípticos, las propiedades se definen como arcos etiquetados y dirigidos, y los valores literales se expresan como nodos rectangulares.

La sentencia en el ejemplo anterior se representaría de la siguiente forma:

Supongamos que ahora queremos decir algo más sobre el creador de la página web anterior, como indicar cuál es su nombre y su dirección de correo electrónico. Entonces, el valor de la propiedad "creador" se convierte en una entidad estructurada (un recurso) que tiene a su vez propiedades con valores. Si el creador tiene un URI propio podemos utilizarlo para identificar el recurso en el siguiente diagrama:



Si el creador no tiene un URI propio se utiliza un recurso *anónimo* (un nodo sin nombre en el diagrama) para agrupar las propiedades de la entidad estructurada valor de la propiedad creador.



En [LS99] se define también una sintaxis concreta interpretable por una máquina, que utiliza XML. Por ejemplo, la página web `http://www.foo.com/john.html` se describe en el siguiente documento RDF diciendo que su creador es alguien con un nombre y una dirección de correo electrónico conocidos (dados por medio de una descripción de recurso anidada).

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:s="http://example.org/schema/">
  <rdf:Description about="http://www.foo.com/john.html">
    <s:Creator>
      <rdf:Description about="http://www.foo.com/staff/123">
        <s:Name>John Smith</s:Name>
        <s:Email>john@foo.com</s:Email>
      </rdf:Description>
    </s:Creator>
  </rdf:Description>
</rdf:RDF>
```

Para identificar de forma única las propiedades y su significado, RDF utiliza el mecanimos de *espacios de nombres* de XML [BHL99]. Las parejas ⟨URI del espacio de nombres, nombre local⟩ se eligen de manera que al concatenar las partes se forma el URI del nodo original[1]. El significado en RDF se expresa mediante una referencia a un *esquema*, lo que explicaremos más adelante. Por ejemplo, en el documento RDF anterior la propiedad `Name` se importa de `http://example.org/schema`. Las sentencias en una `Description` se refieren al recurso determinado por el atributo `about` (interpretado como un URI).

A menudo es necesario referirse a una *colección* de recursos. Los *contenedores* RDF se utilizan para guardar estas colecciones de recursos o literales. Estos contenedores son, a su vez, recursos. RDF define tres tipos de contenedores de objetos: *bolsas* (multiconjuntos o listas no ordenadas de recursos o literales), *secuencias* (listas ordenadas), y *alternativas* (listas que representan alternativas para el valor de una propiedad).

El siguiente ejemplo ilustra el uso de los contenedores: los trabajos de un autor se ordenan en dos secuencias, primero por fecha de publicación y después alfabéticamente por tema.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Seq ID="PapersByDate">
    <rdf:li resource="http://www.dogworld.com/Aug96.doc"/>
    <rdf:li resource="http://www.webnuts.net/Jan97.html"/>
    <rdf:li resource="http://www.carchat.com/Sept97.html"/>
  </rdf:Seq>
  <rdf:Seq ID="PapersBySubj">
    <rdf:li resource="http://www.carchat.com/Sept97.html"/>
    <rdf:li resource="http://www.dogworld.com/Aug96.doc"/>
    <rdf:li resource="http://www.webnuts.net/Jan97.html"/>
  </rdf:Seq>
</rdf:RDF>
```

Los dos recursos de tipo secuencia que se han creado no son recursos que existan de forma independiente a esta descripción y por eso no se ha utilizado el atributo `about` para

[1] Para simplificar los diagramas anteriores, se han utilizado como nombres de propiedades los nombres locales `Creator`, `Name` y `Email`, y no los nombres completos que serían `http://example.org/schema/Creator`, `http://example.org/schema/Name` y `http://example.org/schema/Email`, respectivamente.

indicar su URI, sino el atributo `ID` para darles un identificador único en el documento que contiene esta descripción.

Además de poder expresar sentencias sobre recursos web, RDF puede utilizarse para expresar sentencias acerca de otras sentencias RDF. La sentencia original se modela como un recurso con tres propiedades: sujeto, predicado y objeto. Este proceso se denomina formalmente *reificación* y el modelo de una sentencia se denomina *sentencia reificada*.

El siguiente documento muestra como sentencia reificada la sentencia

"El creador de la página web http://www.foo.com/john.html es John Smith."

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  <rdf:Description>
    <rdf:subject resource="http://www.foo.com/john.html" />
    <rdf:predicate resource="http://example.org/schema/Creator" />
    <rdf:object>John Smith</rdf:object>
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement" />
  </rdf:Description>
</rdf:RDF>
```

Las sentencias reificadas son instancias de la clase `Statement` (véase explicaciones sobre clases en RDF a continuación), por lo que tienen una propiedad `type` con ese valor. Una sentencia y su correspondiente sentencia reificada existen de forma independiente en un documento RDF, y cualquiera puede estar presente sin la otra.

Las comunidades de usuarios de RDF necesitan poder decir ciertas cosas sobre ciertas clases de recursos. La declaración de estas propiedades (atributos) y clases de recursos (clases) se define en un *esquema RDF* [BG02]. Este mecanismo proporciona un *sistema de tipos* básico para ser usado en los documentos RDF. En un esquema RDF, en vez de definir una clase en términos de las propiedades que sus instancias pueden tener, son las propiedades las que se definen en términos de las clases de recursos a las cuales se pueden aplicar.

El siguiente documento RDFS describe una clase de impresoras con una subclase de impresoras láser, y propiedades de las impresoras, como su resolución y su precio:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

  <rdfs:Class ID="Printer">
    <rdfs:comment>The class of printers.</rdfs:comment>
  </rdfs:Class>

  <rdfs:Class ID="Printer">
    <rdfs:subClassOf
      rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  </rdfs:Class>
```

RDF/RDFS	Maude
Documento RDF	Módulo orientado a objetos
Clase	Clase
Recurso	Objeto
Propiedad	Atributo
Contenedor	Tipo abstracto de datos
URI	Identificador de objeto

Tabla 9.1: Conceptos RDF traducidos a Maude.

```
  <rdf:Property ID="PrinterResolution">
    <rdfs:domain rdf:resource="#Printer"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Property>

  <rdf:Property ID="Price">
    <rdfs:domain rdf:resource="#Printer"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Property>
</rdf:RDF>
```

La propiedad `domain` se utiliza para indicar la clase sobre cuyos miembros se puede utilizar la propiedad. La propiedad `range` se utiliza para indicar la clase de la cual tienen que ser miembros los valores de una propiedad.

9.2. Traducción de RDF/RDFS a Maude

Para poder definir la traducción de documentos RDF y RDFS a Maude hemos identificado cuáles son las principales componentes de un documento RDF o RDFS. Estos son los recursos, las propiedades, los contenedores, los URIs y las clases. También hemos identificado qué elementos de un módulo Maude corresponderían de forma natural a estas componentes de RDF. La principal conclusión es que los módulos orientados a objetos en Maude son una buena elección para representar los documentos RDF en Maude, dándoles la semántica precisa deseada. La Tabla 9.1 muestra la correspondencia entre componentes RDF y elementos de Maude.

En esta sección describiremos la traducción de documentos RDF (incluyendo contenedores y reificación) y RDFS a módulos orientados a objetos en Maude. La idea principal es que una descripción RDF de un recurso será traducida a un objeto en Maude.

De la misma manera que hay esquemas RDF que describen los elementos básicos de RDFS y RDF[2], hemos especificado en Maude módulos que describen estos elementos o vo-

[2]Los documentos RDF en los ficheros con URIs http://www.w3.org/2000/01/rdf-schema y http://www.w3.org/1999/02/22-rdf-syntax-ns.

cabulario básicos. Estos módulos serán incluidos en la traducción de cualquier documento RDF concreto que utilice el vocabulario predefinido.

El siguiente módulo define el vocabulario básico de RDFS. Se declara un tipo de referencias URI y se indica que estas pueden utilizarse como identificadores de objeto (`Oid`). Se define una clase de recursos con varios atributos. Todo recurso descrito será una instancia de esta clase, aunque hemos utilizado la misma idea relajada presente en RDF de lo que es una instancia. Un objeto O es una instancia de una clase C si se declara como perteneciente a esta clase y solo tiene atributos definidos para esta clase (o cualquiera de sus superclases), pero no todos los atributos de la clase tienen que estar definidos para los objetos de la clase. Las referencias URI y las instancias de la clase de los recursos se incluyen juntos en un tipo más general `Resource`. También se define una clase para las propiedades, y se declara como subclase de los recursos. Finalmente, hay un tipo de datos para representar los literales, que utiliza el tipo predefinido de identificadores con comilla (`Qid`).

```
(omod http://www.w3.org/2000/01/rdf-schema is
  including QID .

  sorts URI Resource .

  op uri : Qid -> URI .
  subsort URI < Oid .

  class ResourceClass | comment : Literal,
                        label : Literal,
                        seeAlso : Resource,
                        isDefinedBy : Resource .

  subsorts URI ResourceClass < Resource .

  class Property .
  subclass Property < ResourceClass .

  sort Literal .
  op literal : Qid -> Literal .
endom)
```

También hemos definido un módulo con el vocabulario predefinido de RDF. Este declara una clase `Statement` para representar las sentencias RDF, de modo que una sentencia reificada se representará como una instancia de esta clase. La clase tiene tres atributos: `subject`, `predicate` y `object`. El módulo también declara clases para los diferentes contenedores RDF dando definiciones precisas de lo que realmente significan. Por ejemplo, existe una clase para los contenedores de tipo *bolsa* que se describen en [LS99] como "listas no ordenadas de recursos o literales". En Maude, podemos definir qué significa esto exactamente definiendo un tipo de datos para *multiconjuntos* de recursos y literales, con un operador constante `mtBag` para representar el multiconjunto vacío y un operador de unión `_&_` que se declara como asociativo, conmutativo, y con el multiconjunto vacío como

elemento identidad. También hay clases similares para las secuencias y las alternativas, aunque en cada caso el operador de unión se define de una forma diferente. Por ejemplo, el operador de unión para las secuencias se declara como asociativo y con la secuencia vacía como elemento identidad, pero no como conmutativo, porque, por definición, una secuencia es "una lista ordenada de recursos y literales" [LS99].

```
(omod http://www.w3.org/1999/02/22-rdf-syntax-ns is
  including http://www.w3.org/2000/01/rdf-schema .

  class Statement | subject : Resource,
                    predicate : Property,
                    object : Resource .

  *** containers
  class Container .
  subclass Container < Resource .

  class Bag | val : BagVal .
  subclass Bag < Container .
  sort BagVal .
  subsorts Literal Resource < BagVal .
  op mtBag : -> BagVal .
  op _&_ : BagVal BagVal -> BagVal [assoc comm id: mtBag] .

  class Seq | val : SeqVal .
  subclass Seq < Container .
  sort SeqVal .
  subsorts Literal Resource < SeqVal .
  op mtSeq : -> SeqVal .
  op _,_ : SeqVal SeqVal -> SeqVal [assoc id: mtSeq] .

  ...
endom)
```

La traducción de un documento RDF definido por el usuario en un módulo orientado a objetos en Maude se resume en la Tabla 9.1.

A continuación describimos por medio de ejemplos cómo se traducen los documentos RDF definidos por el usuario. El documento RDF que describe la página web `http://www.foo.com/john.html` (Sección 9.1) se traduce al siguiente módulo orientado a objetos en Maude:

```
omod example is
  including http://www.w3.org/1999/02/22-rdf-syntax-ns .
  including http://example.org/schema .

  op http://www.foo.com/john.html : -> Object .
  eq http://www.foo.com/john.html =
     < uri('http://www.foo.com/john.html) : ResourceClass |
       Creator : uri('http://www.foo.com/staff/123) > .
```

```
  op http://www.foo.com/staff/123 : -> Object .
  eq http://www.foo.com/staff/123 =
     < uri('http://www.foo.com/staff/123) : ResourceClass |
       Name : literal('John`Smith),
       Email : literal('john@foo.com) > .
endom
```

Los dos espacios de nombres utilizados en el documento RDF han sido traducidos a inclusiones de módulos. Los dos recursos, uno de ellos anidado dentro del otro, se han traducido a dos objetos constantes y a dos ecuaciones que los definen. El primer objeto tiene un atributo "creador", traducción de la correspondiente propiedad en la descripción RDF, cuyo valor es el URI del otro objeto. De esta forma existe un enlace explícito entre el primer objeto y el segundo. El segundo objeto tiene dos atributos cuyos valores son literales.

Los recursos *anónimos* también son soportados y traducidos a objetos constantes como los anteriores, aunque para nombrarlos en vez de utilizar un URI se utiliza un identificador local. Las descripciones de los contenedores se traducen a objetos de una clase como la clase `Seq` mostrada anteriormente, y los elementos enumerados se incluyen en su atributo como un valor construido utilizando el correspondiente operador de unión. Por ejemplo, el documento RDF que describe los trabajos de un autor, ordenados de las dos formas diferentes que indicamos en la Sección 9.1, se traduce de la siguiente manera:

```
omod DocsCollections is
  including http://www.w3.org/1999/02/22-rdf-syntax-ns .

  op PapersByDate : -> Object .
  eq PapersByDate = < local('PapersByDate) : Seq |
                      val : uri('http://www.dogworld.com/Aug96.doc) ,
                            uri('http://www.webnuts.net/Jan97.html) ,
                            uri('http://carchat.com/Sept97.html) > .

  op PapersBySubj : -> Object .
  eq PapersBySubj = < local('PapersBySubj) : Seq |
                      val : uri('http://carchat.com/Sept97.html) ,
                            uri('http://www.dogworld.com/Aug96.doc) ,
                            uri('http://www.webnuts.net/Jan97.html) > .

endom
```

El documento RDFS con el vocabulario para describir impresoras (Sección 9.1) se traduce de la siguiente manera:

```
omod Printers is
  including http://www.w3.org/1999/02/22-rdf-syntax-ns .
  including http://www.w3.org/2000/01/rdf-schema .

  class Printer | PrinterResolution : Literal, Price : Literal .
```

```
  subclass Printer < ResourceClass .

  class LaserPrinter .
  subclass LaserPrinter < Printer .
endom
```

Como antes, los espacios de nombres se han traducido a inclusiones de módulos. Las declaraciones de clases en RDFS se han traducido a declaraciones de clases en Maude, y las propiedades de subclases se han traducido en declaraciones de subclases en el módulo Maude. Además la clase `Printer` se declara como subclase de la superclase `ResourceClass` que engloba todos los recursos web. La propiedad `PrinterResolution`, con dominio `Printer` y rango `Literal`, se ha traducido a una declaración de atributo de la clase `Printer` cuyos valores pueden ser de tipo `Literal`. La propiedad `Price` se ha tratado de manera completamente análoga.

9.3. Traducción automática definida en Maude

Utilizando las propiedades reflexivas de la lógica de reescritura y Maude, y subiendo al metanivel, podemos definir operaciones que realizan la traducción descrita en la sección anterior de forma automática.

La operación

```
  op translate : Qid RDF -> Module .
```

recibe un identificador con comilla que representa el URI del fichero con la descripción RDF, que será utilizado para nombrar el módulo que se devuelve como resultado, y un valor de tipo RDF. Este valor representa una descripción RDF o RDFS, aunque con una sintaxis un poco diferente a la de XML utilizada en la sección previa, que puede ser leida por un intérprete Maude. La operación `translate`, definida por medio de ecuaciones en un módulo funcional Maude, devuelve el módulo correspondiente a la descripción RDF dada.

A continuación, explicaremos cómo se realiza esta traducción y mostraremos algunas de las ecuaciones que la definen.

La operación `tranlate` recorre los componentes del valor `RDF` y construye paso a paso el módulo que incluye la traducción de cada componente, como se explicó a través de ejemplos en la sección anterior.

```
  vars Q Q' : Qid .
  var NSDL : NSDeclList .
  var DL : DescriptionList .

  eq translate(Q, <rdf:RDF NSDL > DL </rdf:RDF>) =
     translate(DL, addNS(NSDL, setName(emptyMod, Q))) .
```

Si se encuentra una declaración de espacio de nombres, se añade una declaración `including` que incluye un módulo cuyo nombre es el URI del espacio de nombres.

```
 var M : Module .

 eq addNS(xmlns: Q=Q' NSDL, M) =
    addNS(NSDL, addImportList((including Q' .), M)) .
```

Si se encuentra una descripción de recurso, se añade un nuevo operador constante que declara un objeto cuyo nombre es el URI del recurso, y se añade una ecuación definiendo su valor inicial.

```
var IAA : IdAboutAttr .
vars PEL PEL' : PropertyEltList .

eq translate(<rdf:Description IAA > PEL </rdf:Description> DL, M) =
   translate(DL,
    translate(name(IAA), 'ResourceClass, PEL, {'none}'AttributeSet,
      addOpDeclSet(op name(IAA) : -> 'Object ., M))) .

eq translate(< Q:Q' IAA > PEL </ Q:Q' > DL, M) =
   translate(DL,
    translate(name(IAA), Q', PEL, {'none}'AttributeSet,
     addOpDeclSet(op name(IAA) : -> 'Object ., M))) .
```

En estas ecuaciones, se indica la clase del recurso (si aparece en la descripción del recurso; en otro caso se utiliza la clase general `ResourceClass`), y sus atributos con sus valores, que son extraidos de la descripción del recurso. Si el valor de una propiedad P es una descripción de un recurso anidado, posiblemente anónimo, la traducción del recurso anidado se añade de forma recursiva al módulo que se está construyendo, y su URI se utiliza como valor del atributo traducido P. Los contenedores se traducen de una forma similar.

```
var IA : IdAttr .
var ML : MemberList .

eq translate(<rdf:Seq IA > ML </rdf:Seq> DL, M) =
   translate(DL,
    translateSeq(name(IA), ML, {'mtSeq}'SeqVal,
     addOpDeclSet(op name(IA) : -> 'Object ., M))) .
```

Si encontramos un recurso de clase, se incluye en el módulo una declaración de clase, y si la clase tiene propiedades `subClassOf`, estas se incluyen como declaraciones de subclase.

```
eq translate(<rdfs:Class IA > PEL </rdfs:Class> DL, M) =
    translate(DL,
     translate(name(IA), PEL,
      addSubsortDeclSet(subsort 'URI < name(IA) .,
       addClass(name(IA), M)))) .
eq translate(Q,
    <rdfs:subClassOf resource= Q' /> PEL, M) =
     translate(Q, PEL,
      addSubclassDecl(subclass Q < Q' ., M)) .
```

Finalmente, si encontramos una descripción de propiedad, esta se traduce como un atributo del tipo especificado en la propiedad `range` de la clase especificada en la propiedad `domain`.

En la Sección 9.5 utilizaremos esta traducción en una aplicación de compra de impresoras en un entorno móvil, implementada utilizando la extensión de Maude para permitir cómputos móviles conocida como Mobile Maude, que describimos a continuación.

9.4. Mobile Maude

La flexibilidad de la lógica de reescritura para representar muchos y variados estilos de comunicación, tanto síncrona como asíncrona, su facilidad para soportar objetos distribuidos y concurrentes, y su capacidad reflexiva para soportar la metaprogramación y la reconfiguración dinámica, la convierten en un formalismo único para la especificación de sistemas distribuidos basados en agentes móviles, sobre las cuales se podrá basar la demostración de propiedades de seguridad, corrección, y rendimiento. Es más, la disponibilidad de una implementación de la lógica de reescritura, vía Maude, convierte tales especificaciones en ejecutables proporcionando una excelente herramienta de prototipado en la que ejecutar las aplicaciones basadas en agentes móviles.

En [DELM00] se ha propuesto una extensión del lenguaje Maude con mecanismos para soportar la computación móvil, conocida como Mobile Maude. Aquí se presenta una breve descripción del lenguaje, mostrando sus conceptos más importantes y sus primitivas.

Mobile Maude es un lenguaje de agentes móviles que extiende al lenguaje Maude para soportar cómputos móviles. En su diseño se ha hecho un uso sistemático de la reflexión, obteniéndose un lenguaje de agentes móviles declarativo, simple y general. Mobile Maude ha sido especificado formalmente a través de una teoría de reescritura en Maude. Ya que esta especificación es ejecutable, puede utilizarse como prototipo del lenguaje, en el cual se pueden simular sistemas de agentes móviles.

Las principales características del lenguaje Mobile Maude, que lo hacen adecuado para los objetivos asociados a la especificación y prototipado de sistemas basados en agentes móviles, son las siguientes:

- Basado en la lógica de reescritura, una lógica simple y sencilla de usar para la especificación de sistemas concurrentes, distribuidos y abiertos, incluyendo los datos, los eventos, la evolución del sistema, la seguridad, y aspectos de tiempo real.

- Orientado a objetos, con soporte de primer nivel para objetos y agentes distribuidos y su comunicación, tanto síncrona como asíncrona, sin perder la semántica formal.

- Reflexivo, con soporte formal para la metaprogramación a cualquier nivel, y para la reconfiguración dinámica con gran flexibilidad.

- De amplio espectro, permitiendo la transformación y refinamiento rigurosos desde especificaciones hasta programas declarativos.

- Modular, con un avanzado sistema de módulos que soportan parametrización, teorías para requisitos, vistas, y expresiones de módulos, con sus correspondientes operaciones de instanciación, composición y aplanamiento.
- Móvil, con soporte para el movimiento de datos, estados locales, y programas.
- Con una base formal para el desarrollo de modelos de seguridad y la verificación de propiedades sobre tales modelos.

Mobile Maude está basado en dos conceptos principales: *procesos* y *objetos móviles*. Los procesos son entornos computacionales localizados donde pueden residir los objetos móviles. Los objetos móviles pueden moverse entre diferentes procesos, pueden comunicarse de forma asíncrona con otros objetos por medio de mensajes, y pueden evolucionar, por medio de pasos de ejecución. En Mobile Maude, un objeto móvil puede comunicarse con objetos en el mismo proceso o en procesos diferentes[3].

Los objetos móviles viajan con sus propios datos y código. El código de un objeto móvil viene dado por (la metarrepresentación de) un módulo orientado a objetos (una teoría de reescritura) y sus datos vienen dados por una configuración de objetos y mensajes que representan su estado. Tal configuración es un término válido en el módulo del código, el cual se utiliza para ejecutarla.

Los procesos y los objetos móviles se modelan como objetos distribuidos de las clases `P` y `MO`, respectivamente. Los nombres de los procesos son del tipo `Pid`, y la declaración de la clase `P` de procesos es la siguiente:

```
class P | cf : Configuration, cnt : Int, guests : Set(Mid),
          forward : PFun(Int, Tuple(Pid, Int)) .
```

El atributo `cf` representa la configuración *interna*, donde puede haber objetos móviles y mensajes. Por tanto, un sistema de agentes móviles consistirá en una configuración de procesos y mensajes en tránsito entre procesos, con una configuración de objetos móviles dentro de cada proceso, como se muestra en la Figura 9.2. El atributo `guests` guarda los nombres de los objetos móviles que actualmente se encuentran en la configuración interna del proceso. El atributo `cnt` es un contador para generar nuevos nombres de objetos móviles. Estos nombres de objetos móviles son del tipo `Mid`, y tienen la forma `o(PI, N)`, donde `PI` es el nombre del proceso padre, donde fue creado el objeto, y `N` es un número que distingue a los hijos del proceso `PI`. Este número lo proporciona el atributo `cnt`, que se incrementa tras la creación de un nuevo objeto móvil. El reparto de mensajes no es trivial, ya que los objetos móviles pueden moverse de un proceso a otro. Para resolver este problema, cada proceso guarda información sobre el lugar donde se encuentran sus hijos en el atributo `forward`, una función parcial que hace corresponder a cada número de hijo un par formado por el nombre del proceso donde el objeto está actualmente, y el número de "saltos" que el objeto ha dado hasta llegar allí. El número de saltos se utiliza para conocer la antigüedad de la información mantenida por el padre. Cada vez que un objeto

[3] Algunos lenguajes de agentes móviles, como el Ambient Calculus [Car99], prohíben este último tipo de comunicación, permitiendo únicamente la comunicación dentro de un mismo proceso.

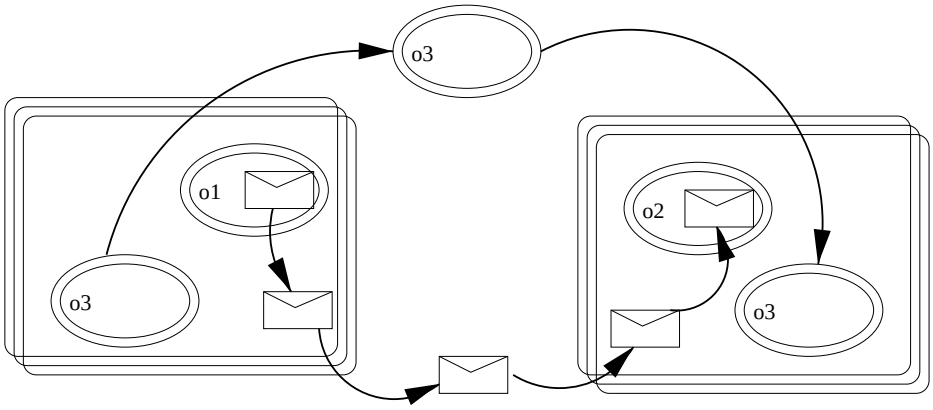


Figura 9.2: Posible configuración en Mobile Maude.

móvil se mueve a un proceso diferente, envía un mensaje a su padre anunciando la nueva localización.

La clase de objetos móviles está declarada de la siguiente manera:

```
class MO | mod : Module, s : Term, p : Pid, hops : Int, mode : Mode .
```

El atributo `s` mantiene (la metarrepresentación de) el estado del objeto móvil, y tiene que ser de la forma `C & C'`, donde `C` es una configuración de objetos y mensajes (mensajes de entrada aún no procesados y mensajes entre objetos internos) y `C'` es un multiconjunto de mensajes (la bandeja de mensajes *de salida*). Uno de los objetos en la configuración de objetos y mensajes se supone que tiene el mismo identificador que el objeto móvil en el que se encuentra. A este objeto se le denomina *objeto principal*. El atributo `mod` es la metarrepresentación del módulo Maude que describe el comportamiento del objeto. El atributo `p` tiene el nombre del proceso donde el objeto se encuentra actualmente. El número de saltos de un proceso a otro realizados por el objeto se almacena en el atributo `hops`. Finalmente, el atribujo `mode` de un objeto puede ser `active` si el objeto se encuentra dentro de un proceso (y por tanto puede ejecutarse), o `idle` si el objeto está en movimiento viajando de un proceso a otro.

Como dijimos anteriormente, la semántica de Mobile Maude está especificada por medio de una teoría de reescritura orientada a objetos que contiene las definiciones de las clases anteriores y reglas de reescritura que describen el comportamiento de las diferentes primitivas: movilidad de objetos, paso de mensajes, y creación de objetos y procesos. Esta especificación es el *código de sistema* de Mobile Maude, el cual se puede utilizar como prototipo sobre el cual ejecutar aplicaciones Mobile Maude, como haremos en la Sección 9.5. Por supuesto, tales aplicaciones tienen que satisfacer ciertos requisitos, como ser orientadas a objetos, utilizar el constructor `_&_` para enviar mensajes fuera del objeto móvil, y utilizar los mensajes predefinidos para moverse a otros procesos.

A continuación, presentaremos las primitivas de Mobile Maude, junto con algunas reglas de reescritura que definen su comportamiento.

Existen tres clases de comunicación entre objetos. Los objetos dentro de un mismo objeto móvil pueden comunicarse entre sí por medio de mensajes con cualquier formato, y la comunicación puede ser síncrona o asíncrona. Los objetos en diferentes objetos móviles pueden comunicarse cuando están en el mismo proceso o cuando están en procesos diferentes; en estos casos, la clase de comunicación utilizada es transparente a los objetos que se comunican, pero ha de tratarse de comunicaciones asíncronas, y los mensajes tienen que ser de la forma `to_:_`, donde el primer argumento es el identificador del objeto destino, y el segundo argumento es el contenido del mensaje, un valor del tipo `Contents` construido con una sintaxis libremente definida por el usuario (véase Sección 9.5). De modo que la información mínima necesaria para enviar un mensaje es la identidad del receptor; si el emisor quiere comunicar su identidad, deberá incluirla en el contenido del mensaje. Si el receptor es un objeto en otro objeto móvil, entonces el mensaje debe ser colocado por el emisor en la segunda componente de su estado (la bandeja de mensajes de salida). El código de sistema enviará el mensaje al objeto destino.

A continuación, mostramos algunas de las reglas encargadas del reparto de mensajes. La primera de ellas, en la que los términos aparecen metarrepresentados (por ejemplo, el término `T'` metarrepresenta el nombre del objeto destino), inicia el proceso sacando el mensaje del objeto móvil:

```
rl [message-out-to] :
  < M : MO | mod : MOD, s : '_&_[T, 'to_:_[T', T'']], mode : active >
  => < M : MO | mod : MOD, s : '_&_[T, 'none.MsgSet] >
     (to downMid(T') { T'' }) .
```

Si el mensaje se envía a un objeto móvil `o(PI', N)` en un proceso diferente, debe enviarse al proceso padre `PI'` el cual lo redirigirá apropiadamente utilizando la información en su atributo `forward`. Obsérvese como la condición de la siguiente regla comprueba que `o(PI', N)` no está en el proceso `PI` mirando en el conjunto de huéspedes.

```
crl [msg-send] :
  < PI : P | cf : C (to o(PI', N) { T }), guests : SMO, forward : F >
  => < PI : P | cf : C >
     to o(PI', N) hops null in PI' { T }
  if (not o(PI', N) in SMO) and PI =/= PI' .
```

Cuando el mensaje llega al proceso donde se encuentra el objeto destino, primero se introduce en la configuración interna del proceso, y finalmente en el estado interno del objeto móvil destinatario (aquí metarrepresentado).

```
crl [msg-arrive-to-proc] :
  to o(PI, N) hops H in PI' { T' }
  < PI' : P | cf : C, guests : SMO >
  => < PI' : P | cf : C (to o(PI, N) { T' }) >
  if o(PI, N) in SMO .
```

```
rl [msg-in] :
  to M { T }
  < M : MO | mod : MOD, s : '_&_[T', T''] >
  => < M : MO | s : '_&_['__['to_:_[up(M), T], T'], T''] > .
```

Cuando un objeto móvil quiere moverse a otro proceso debe poner en su bandeja de mensajes de salida un mensaje del tipo `go(PI)`, donde `PI` es el identificador del proceso destino. Cuando un objeto móvil tiene un mensaje `go` en la bandeja de salida, se envía un mensaje `go` entre procesos, con el objeto móvil como uno de sus argumentos, después de haber eliminado el mensaje de la bandeja de salida y haber puesto el estado del objeto móvil a `idle`. Esto asegura que el objeto móvil estará inactivo mientras esté en movimiento.

```
rl [message-out-move] :
  < M : MO | s : '_&_[T, 'go[T']], mode : active >
  => go(downPid(T'), < M : MO | s : '_&_[T, 'none.MsgSet], mode : idle >) .
```

Si el emisor del mensaje y el destino están en diferentes procesos, entonces este mensaje `go` tiene que viajar al proceso deseado, saliendo a la configuración externa de procesos. Cuando el mensaje alcanza el proceso destino, el objeto móvil es introducido en este proceso (en modo `active`), y el proceso padre es informado, para que pueda actualizar su información en el atributo `forward`.

```
rl [arrive-proc] :
  go(PI, < o(PI', N) : MO | hops : N' >)
  < PI : P | cf : C, guests : SMO, forward : F >
  => if PI == PI'
     then < PI : P | cf : C < o(PI', N) : MO |
                               p : PI, hops : N' + 1, mode : active >,
                     guests : o(PI', N) . SMO,
                     forward : F[N -> (PI, N' + 1)] >
     else < PI : P | cf : C < o(PI', N) : MO |
                               p : PI, hops : N' + 1, mode : active >,
                     guests : o(PI', N) . SMO >
          (to PI' @ (PI, N' + 1) { N })
     fi .
```

En el mensaje `go`, el objeto móvil indica el proceso a donde quiere ir. En ocasiones, un objeto móvil querrá ir a donde se encuentre otro objeto móvil, pero solo conocerá en principio el identificador del objeto con el que se quiere encontrar, y no el proceso en el que este se encuentra. En tal caso, se utilizará el mensaje predefinido `go-find`. Cuando este mensaje es utilizado por un objeto `M`, toma como argumentos el identificador del objeto móvil al que `M` quiere alcanzar, y el identificador de un proceso donde el destino podría encontrarse.

Cuando un objeto quiere crear un nuevo objeto móvil, debe enviar un mensaje `newo` al sistema, colocándolo en la segunda componente de su estado. El mensaje `newo` toma como argumentos (la metarrepresentación de) un módulo M, una configuración C (que será la configuración inicial colocada dentro del objeto móvil a crear, y que tiene que ser

un término válido del módulo M), y el identificador provisional del objeto principal en la configuración C. La primera acción realizada por el sistema cuando se detecta un mensaje `newo` es crear un nuevo objeto móvil con la configuración C como estado y el módulo M como su código, tras lo cual se enviará un mensaje `start-up` al objeto principal con su nuevo nombre, para que coincida con el nombre del objeto móvil en el que se encuentra.

La ejecución de objetos móviles se basa en la reflexión y se lleva a cabo mediante la siguiente regla:

```
rl [do-something] :
  < M : MO | mod : MOD, s : T, mode : active >
  => < M : MO | s : meta-rewrite(MOD, T, 1) > .
```

El código de sistema completo de Mobile Maude, junto con información relacionada, puede encontrarse en http://maude.cs.uiuc.edu/mobile-maude.

El código que describe el comportamiento de los objetos móviles se denomina *código de aplicación*. En la siguiente sección veremos ejemplos de este tipo de código.

En [DV02] hemos presentado un caso de estudio en el que Mobile Maude se utiliza para implementar de forma satisfactoria una aplicación distribuida, a saber, un sistema de revisión de artículos de una conferencia, desde que esta se anuncia hasta que se editan las actas con los artículos aceptados. Este ejemplo fue propuesto por Cardelli [Car99] como reto para cualquier lenguaje para entornos de redes de ámbito global (en inglés, *wide area networks*), para demostrar su usabilidad (aunque el ejemplo ya había sido utilizado anteriormente por otros autores). Esto nos permitió identificar posibles deficiencias y experimentar con diferentes alternativas. Por ejemplo, la primitiva `go-find` antes descrita no estaba en el conjunto original de mensajes del sistema.

9.5. Caso de estudio: compra de impresoras

En esta sección presentamos una aplicación del proceso de traducción de RDF a Maude. La traducción propuesta se utiliza en un ejemplo el que un agente comprador visita a varios vendedores que le dan información sobre las impresoras que venden en formato RDF. El comprador mantiene el precio de la impresora más barata. Este ejemplo se ha implementado utilizando Mobile Maude.

En esta aplicación tenemos dos clases diferentes de objetos móviles: los *vendedores* y los *compradores*. Aunque en esta versión los vendedores no se mueven, también tienen que ser objetos móviles de Mobile Maude, pues se comunican con otros objetos móviles. Hay otra clase de objetos, los objetos *comparadores*, que son utilizados por los compradores para comparar impresoras. Estos no son objetos móviles, como veremos más adelante.

Un comprador visita a varios vendedores. El comprador pide a cada uno la descripción de la impresora que vende. El vendedor envía al comprador esta información en formato RDF, la cual es traducida por el comprador y dada al comparador, que mantiene el precio de la impresora más barata.

Primero definimos los vendedores. Estos son agentes estáticos cuyo comportamiento viene definido por el módulo siguiente. La clase `Seller` tiene un atributo `description` con

la descripción en RDF de la impresora que el vendedor vende, utilizando el vocabulario del esquema RDFS presentado en la Sección 9.1. Cuando el vendedor recibe una petición de esta descripción, la envía en formato RDF.

```
(omod SELLER is
  including RDF-SYNTAX .

  class Seller | description : RDF .

  op get-printer-description : Oid -> Contents .
  op printer-description : RDF -> Contents .

  vars S B : Oid .  var D : RDF .

  rl [get-des] :
      (to S : get-printer-description(B))
      < S : Seller | description : D > & none
   => < S : Seller | > & (to B : printer-description(D)) .
endom)
```

Obsérvese cómo el estado del comprador se describe en la regla `get-des` utilizando el operador `_&_` para así separar el estado interno y los mensajes de entrada de los mensajes de salida. Gracias a ello este módulo se puede utilizar para crear objetos móviles en Mobile Maude.

Antes de definir a los compradores, definimos la clase `Comparer` cuyas instancias son capaces de comparar diferentes impresoras, manteniendo el precio de la más barata. Cuando un objeto comparador está próximo a un objeto impresora, generado por la traducción de una descripción RDF de una impresora, el comparador consulta el precio de la impresora y lo compara con el mejor precio que conoce, actualizando su conocimiento cuando ello sea preciso. Obsérvese que el objeto impresora desaparece, ya que no representa a una impresora real, sino solo a su información, de modo que deja de ser útil cuando el comparador ya ha extraído la información necesaria. El módulo paramétrico `DEFAULT` utilizado extiende el tipo de la vista dada como parámetro con un nuevo valor `null` que representa un valor no definido.

```
(omod COMPARER is
  including Printers .
  including DEFAULT[MachineInt] .

  class Comparer | best : Default[MachineInt] .

  var P C : Oid .
  var Q : Qid .
  var N : MachineInt .
  var Atts : AttributeSet .

  rl [compare] :
     < P : Printer | Price : literal(Q), Atts >
```

```
     < C : Comparer | best : null >
  => < C : Comparer | best : convert(Q) > .

  rl [compare] :
     < P : Printer | Price : literal(Q), Atts >
     < C : Comparer | best : N >
  => < C : Comparer | best : if (convert(Q) < N) then
                             convert(Q) else N fi > .
endom)
```

Un comparador no es un objeto móvil de Mobile Maude. No se mueve de forma independiente, y no puede enviar o recibir mensajes de otro objeto móvil. Es un objeto Maude que viajará dentro de un atributo de un comprador, como veremos más adelante.

Obsérvese cómo se utiliza la variable `Atts` de tipo `AttributeSet` en el objeto impresora. Utilizando esta variable, la regla puede aplicarse a cualquier impresora con *al menos* un atributo `Price`; si la impresora tiene más atributos, estos serán capturados por la variable `Atts`.

Este estilo de programación es muy útil en el ámbito de la web semántica. Si un vendedor define su propio esquema RDFS para sus impresoras, extendiendo el esquema presentado en la Sección 9.1 definiendo una subclase de las impresoras con nuevas propiedades que sean importantes para este vendedor, entonces enviará descripciones de impresoras con algunas propiedades desconocidas por el comparador. Sin embargo, la implementación anterior seguiría siendo útil, ya que las propiedades (atributos) extra serían capturados por la variable `Atts`.

Finalmente, definimos los compradores. El módulo `BUYER` describe el comportamiento de un agente comprador. Este tiene una lista `IPs` con las direcciones de los vendedores que conoce y quiere visitar. Un comprador tiene que visitar a todos los vendedores de su lista, pidiendo a cada uno la descripción de sus impresoras. El comprador tiene un atributo `app-state` con el estado actual del comparador, metarrepresentado. Tiene que estar metarrepresentado porque el comprador quiere ser capaz de ejecutar el comparador, para lo que han de estar a diferentes niveles de representación. Cada vez que recibe una nueva descripción, el comprador traduce la descripción RDF de la impresora a un módulo Maude M con un objeto de la clase `Printer`, coloca este objeto junto con el estado actual de su comparador, y pide que se reescriban (utilizando la operación `meta-rewrite`) en el módulo Maude obtenido uniendo M con el módulo `COMPARER` que contiene el código del comparador. El módulo paramétrico `LIST` se utiliza aquí para crear listas de identificadores de objetos.

```
(omod BUYER is
  including RDF-Translation .
  including LIST[Oid]

  sort Status .
  ops onArrival asking done : -> Status .

  class Buyer | IPs : List[Oid], status : Status, app-state : Term .
```

```
  op get-printer-description : Oid -> Contents .
  op printer-description : RDF -> Contents .

  var PD : RDF .
  var Ss : List[Oid] .
  vars B S : Oid .
  var PI : Pid .
  var N : MachineInt .
  var T : Term .

  rl [move] :
     < B : Buyer | IPs : o(PI,N) + Ss, status : done > & none
  => < B : Buyer | status : onArrival > & go-find(o(PI,N), PI) .

  rl [onArrival] :
     < B : Buyer | IPs : S + Ss, status : onArrival > & none
  => < B : Buyer | status : asking > & (to S : get-printer-description(B)) .

  rl [new-des] :
      (to B : printer-description(PD))
      < B : Buyer | IPs : S + Ss, app-state : T, status : asking >
   => < B : Buyer | IPs : Ss,
                    app-state : meta-rewrite(
                      addDecls(up(COMPARER), translate('Printer, PD)),
                        '__[T, extractResources(translate('Printer, PD))], 0),
                    status : done > .
endom)
```

La primera regla de reescritura, `move`, se encarga de los viajes del comprador: si se ha terminado en el proceso actual (su estado es `done`) y hay al menos un nombre de vendedor en el atributo `IPs`, entonces se pide al sistema que lleve al comprador a donde se encuentra el siguiente vendedor. Al llegar, el comprador pide al vendedor la descripción de su impresora, dando también el nombre del comprador (regla de reescritura `onArrival`). Cuando llega la descripción RDF, el comprador la traduce a Maude (utilizando la operación `translate` descrita en la Sección 9.2), extrae el recurso correspondiente a la descripción de la impresora, lo coloca junto con el comparador (utilizando el operador de unión de configuraciones de objetos Maude `__`), y reescribe el resultado en el módulo con el comportamiento del comparador, lo que cambiará el estado del comparador.

El siguiente módulo `EXAMPLE` define la configuración inicial de un sistema con dos vendedores con sus correspondientes descripciones de impresoras[4] y un comprador. El módulo `EXAMPLE` incluye el código de sistema de Mobile Maude, en el módulo `MOBILE-MAUDE`.

[4]Las descripciones se dan en RDF con la sintaxis que puede entender Maude. Para simplificar la presentación, aquí solo ponemos el precio de cada impresora como propiedad de las mismas. En la página web puede encontrarse un ejemplo más completo.

```
(omod EXAMPLE is
  protecting MOBILE-MAUDE .

  rl [initial-state] : initial =>
   < 'loc0 : R | cnt : 4 >
   < p('loc0,0) : P |
     cnt : 1,
     cf : < o(p('loc0,0),0) : MO |
            mod : up(SELLER),
            s : up(SELLER,
                < o(p('loc0,0),0) : Seller |
                  description :
                  <rdf:RDF
                    xmlns: s 'rdf = s 'http://www.w3.org/1999/02/22-rdf-syntax-ns
                    xmlns: s 'sc = s 'http://printers.org/schema/ >
                    < s 'sc : s 'Printer about= s 'http://HP/HPLaserJet1100 >
                      < s 'sc : s 'Price > s '389 </ s 'sc : s 'Price >
                    </ s 'sc : s 'Printer >
                  </rdf:RDF> > & none),
             p : p('loc0,0),
             hops : 0,
             mode : active >,
     guests : o(p('loc0,0),0),
     forward : (0, (p('loc0, 0), 0)) >

   < p('loc0,1) : P |
     cnt : 1,
     cf : < o(p('loc0,1),0) : MO |
            mod : up(SELLER),
            s : up(SELLER,
                < o(p('loc0,1),0) : Seller |
                  description :
                  <rdf:RDF
                    xmlns: s 'rdf = s 'http://www.w3.org/1999/02/22-rdf-syntax-ns
                    xmlns: s 'sc = s 'http://printers.org/schema/ >
                    < s 'sc : s 'Printer about= s 'http://HP/HPLaserJet1100 >
                      < s 'sc : s 'Price > s '209 </ s 'sc : s 'Price >
                    </ s 'sc : s 'Printer >
                  </rdf:RDF> > & none),
             p : p('loc0,1),
             hops : 0,
             mode : active >,
     guests : o(p('loc0,1),0),
     forward : (0, (p('loc0, 1), 0)) >

   < p('loc0,3) : P |
     cnt : 1,
     cf : < o(p('loc0,3),0) : MO |
            mod : up(BUYER),
            s : up(BUYER,
```

```
                < o(p('loc0,3),0) : Buyer |
                  IPs : (o(p('loc0,0),0) + o(p('loc0,1),0)),
                  status : done,
                  app-state : up(COMPARER, < 'comp : Comparer | best : null >) >
                & none),
            p : p('loc0,3),
            hops : 0,
            mode : active >,
     guests : o(p('loc0,3),0),
     forward : (0, (p('loc0, 3), 0)) > .
endom)
```

Podemos ejecutar el ejemplo, para ver que en efecto el comparador del comprador obtiene la información sobre el precio de la impresora más barata. De la salida producida por Maude hemos eliminado las descripciones de las impresoras que tienen los vendedores, pues estos no cambian. Obsérvese cómo el comprador ha terminado en el proceso del segundo vendedor que ha visitado, y cómo se ha modificado la información del atributo `forward` de su padre.

```
Maude> (MO grew [5][10] initial .)

Result Configuration :
< 'loc0 : R | cnt : 4 >
< p('loc0, 0 ) : P |
   cf : < o(p('loc0,0),0 ) : MO |
          mod : up(SELLER),
          s : up(SELLER, < o(p('loc0,0),0) : Seller |
                           description : ... > & none),
          mode : active ,
          hops : 0 ,
          p : p('loc0,0) >,
   forward : (0, (p('loc0,0),0)),
   guests : o(p('loc0,0),0),
   cnt : 1 >

< p('loc0,1) : P |
   cf : (< o(p('loc0,1),0) : MO |
            mod : up(SELLER),
            s : up(SELLER, < o(p('loc0,1),0) : Seller |
                             description : ... > & none),
            mode : active,
            hops : 0 ,
            p : p('loc0,1) >
          < o(p('loc0,3),0) : MO |
            mod : up(BUYER),
            s : up(BUYER,
                < o(p('loc0,3),0) : Buyer |
                  app-state : ( '<_:_|_>[{''comp}'Qid,{'Comparer}'Comparer,
                                        'best':_[{'209}'NzMachineInt]]),
```

```
                status : done,
                IPs : nilList > & none),
          mode : active ,
          hops : 2,
          p : p('loc0,1) > ),
   forward : (0, (p('loc0,1),0)),
   guests : (o(p('loc0,1),0) . o(p('loc0,3),0)),
   cnt : 1 >

< p('loc0,3) : P |
    cf : none,
    forward : (0, (p('loc0,1),2)),
    guests : mt,
    cnt : 1 >
```

9.6. Conclusiones

En este capítulo hemos visto cómo se puede dotar de semántica formal a los documentos RDF mediante su traducción a módulos orientados a objetos en Maude. Los elementos básicos de un documento RDF, como son los recursos, sus propiedades, las clases y los contenedores, tienen una relación natural con los objetos, sus atributos, clases de objetos y tipos abstractos de datos, respectivamente. Por tanto, la traducción permite precisar la idea intuitiva que hay detrás de las descripciones RDF. Además, esta traducción se puede ejecutar, al contrario de la semántica basada en lógica de predicados propuesta en [Hay02], y que comentaremos más abajo.

Gracias a las propiedades reflexivas de Maude hemos podido implementar esta traducción de documentos RDF a módulos orientados a objetos de Maude en el mismo Maude, mediante una función definida al metanivel que recibe como argumento de entrada un documento RDF y devuelve como resultado un módulo orientado a objetos. El módulo resultado se puede manipular y también ejecutar al metanivel.

Esto nos permite definir de nuevo en Maude agentes web que trabajen con descripciones RDF y que utilicen la traducción para poder utilizarlas junto con aplicaciones escritas en Maude al nivel de las descripciones y que utilizan estos datos para obtener la información requerida, como ocurre en el ejemplo de compra de impresoras descrito en la Sección 9.5. Hemos visto también cómo estas aplicaciones pueden integrarse en un entorno distribuido y móvil utilizando Mobile Maude.

En [Hay02] se presenta una semántica de teoría de modelos para RDF y RDFS. La definición semántica traduce un grafo RDF en una expresión lógica "con el mismo significado". Básicamente, cada arista del grafo da lugar a una aserción atómica y el grafo completo se traduce como el cierre existencial de la conjunción de las aserciones generadas para cada una de sus aristas. También se estudia una noción de deducción en RDF. Aunque versiones previas de dicho trabajo no soportaban la reificación o los contenedores (véase la Sección 9.1), la versión actual sí que los contempla [Hay02]. Nuestro enfoque es más pragmático, en el sentido de que traducimos a un lenguaje formal, pero de modo que las traducciones pueden ser ejecutadas. Así combinamos las ventajas de un mundo formal

en el que se pueden verificar formalmente propiedades, con las del desarrollo de prototipos con los que adquirir confianza en nuestras especificaciones e implementaciones de sistemas.

Capítulo 10

Conclusiones y trabajo futuro

Como dijimos en la introducción, hemos abordado el objetivo de presentar la lógica de reescritura, y en particular el lenguaje Maude, como *marco semántico ejecutable* desde diferentes puntos de vista:

- obteniendo semánticas operacionales ejecutables para álgebras de procesos como CCS y LOTOS, así como para lenguajes funcionales e imperativos;
- realizando el modelado y análisis del protocolo de elección de líder en la especificación del bus multimedia en serie IEEE 1394 ("FireWire"); y
- dotando de semántica formal a lenguajes de la web semántica, mediante la traducción de RDF (Resource Description Framework) a Maude y su integración con Mobile Maude.

A continuación presentamos las conclusiones y líneas de trabajo futuro que nos gustaría seguir en relación con estos tres puntos.

10.1. Semánticas operacionales ejecutables

En primer lugar, hemos presentado dos implementaciones diferentes de la semántica de CCS y su utilización para implementar la lógica modal de Hennessy-Milner. Hemos mostrado con todo detalle cómo implementar la semántica de CCS siguiendo el enfoque de las reglas semánticas como reglas de reescritura. Se han resuelto de una forma general los problemas causados por la presencia de nuevas variables en la parte derecha de las reglas y el no determinismo, utilizando para ello las propiedades reflexivas de la lógica de reescritura y su realización en el módulo `META-LEVEL` de Maude. Esto nos permitió utilizar las mismas técnicas para implementar la lógica modal de Hennessy-Milner. Por último comparamos nuestro enfoque con la utilización del demostrador de teoremas Isabelle/HOL, llegando a la conclusión de que, aunque Maude no tenga implementadas todas las facilidades de demostración que ofrece Isabelle, al no ser ese el objetivo de Maude al tratarse de un

marco tan general, sí que gracias a la reflexión puede utilizarse para aplicar técnicas de order superior en un marco de primer orden.

También hemos presentado una implementación alternativa de la semántica de CCS, siguiendo el enfoque de transiciones como reescrituras. Como dijimos en la Sección 5.5, la segunda implementación ofrece diversas ventajas, siendo más cercana a la presentación matemática de la semántica, mientras que el primer enfoque necesita estructuras, como los multiconjuntos de juicios, y mecanismos auxiliares, como la generación de nuevas metavariables y su propagación.

Las técnicas presentadas en el Capítulo 3 y utilizadas en el caso de CCS, son muy generales. Además, el segundo enfoque, en el que las transiciones se representan como reescrituras, es interesante en la práctica, al conducir a implementaciones razonablemente eficientes.

El enfoque de transiciones como reescrituras se ha utilizado también para implementar otros tipos de semánticas operacionales. En concreto, se utiliza para implementar de forma satisfactoria todas las semánticas diferentes presentadas por Hennessy en [Hen90] para lenguajes tanto funcionales como imperativos, así como la semántica de Mini-ML presentada por Kahn [Kah87]. Este mismo enfoque también ha sido utilizado por Thati, Sen y Martí Oliet en [TSMO02] para obtener una especificación ejecutable de una versión asíncrona del π-cálculo.

Por último, hemos descrito una implementación de una semántica simbólica para LOTOS siguiendo también el enfoque de transiciones como reescrituras. Además, las especificaciones en ACT ONE, que pueden formar parte de las especificaciones LOTOS, se traducen automáticamente a módulos funcionales equivalentes en Maude. Estos módulos se extienden para integrarlos con la semántica operacional. El problema que se resuelve con esta extensión es el de obtener operaciones cuya definición natural esté basada en la sintaxis de tipos de datos definibles por el usuario: a la hora de introducir una especificación en ACT ONE se crean automáticamente las definiciones de estas operaciones. Finalmente, hemos mostrado cómo implementar en Maude un interfaz de usuario que permite que este no tenga que utilizar Maude directamente, sino que utilice una herramienta escrita sobre Maude donde puede introducir una especificación LOTOS y ejecutarla mediante comandos específicos que recorren el correspondiente sistema de transiciones etiquetado asignado a la especificación por la semántica simbólica.

También hemos comparado esa implementación de la semántica de LOTOS con otra que sigue el enfoque alternativo de reglas de inferencia como reescrituras. Ya que este lenguaje es más complejo que CCS, con lo que las especificaciones LOTOS son bastante más complicadas, incluyendo también tipos de datos, hemos necesitado mejorar en varias direcciones las soluciones que facilitaron la implementación de CCS.

El estudio de las semánticas operacionales ejecutables en Maude no termina ni mucho menos con esta tesis. A corto plazo pretendemos estudiar un lenguaje de estrategias internas para Maude, aplicable no solo a la representación de semánticas operacionales sino a especificaciones ejecutables en general. Por el momento cada usuario tiene que escribir sus propias estrategias, bien partiendo de cero, o bien adaptando y extendiendo las estrategias propuestas por otros investigadores, como hemos hecho en los Capítulos 4 y 8. Para

solventar esta situación, nos hemos planteado el objetivo de diseñar un lenguaje básico de estrategias para el usuario, integrado en el sistema Maude, que sea suficientemente expresivo para cubrir gran parte de los casos de uso esencial de estrategias que se precisan en la práctica. Las técnicas de reflexión y metaprogramación nos permitirán realizar un prototipo de implementación del lenguaje de estrategias en el propio metanivel de Maude. Una vez este prototipo demuestre la utilidad del lenguaje propuesto, se trasladará al equipo de implementación de Maude, para su integración en la implementación más eficiente escrita en C++.

También estamos interesados en aplicar la metodología presentada en esta tesis a nuevos lenguajes en desarrollo o desarrollados recientemente. Nos interesan particularmente aquellos lenguajes en los que el uso de estrategias de ejecución resulta esencial. Sobre ellos podremos aprovechar, por un lado, los resultados y experiencia obtenidos en esta tesis, y por otro lado el lenguaje de estrategias internas que acabamos de comentar. Como ejemplos específicos, pretendemos estudiar diferentes semánticas ejecutables del lenguaje funcional paralelo Edén [BLOMPM96, HHOM02], sobre las que iríamos variando las estrategias de ejecución para ver cómo influyen en las propiedades de la semántica, y del álgebra de procesos E-LOTOS [ISO01], donde las estrategias jugarían un papel importante a la hora de combinar las transiciones de la semántica dinámica con las transiciones relacionadas con el paso del tiempo.

Basándonos en la semántica simbólica de LOTOS utilizada en el Capítulo 7, se han definido ya una bisimulación simbólica [CS01, RC02] y la lógica modal FULL [CMS01]. Tenemos planeado extender nuestra herramienta de forma que se pueda comprobar si dos procesos son bisimilares, o si un proceso satisface una fórmula de la lógica modal dada. De hecho, ya hemos implementado un subconjunto de FULL sin valores (Sección 7.4), y hemos integrado la extensión con nuestra herramienta. La parte de la lógica con valores requiere un estudio más profundo, y creemos que será inevitable el recurso a algún tipo de demostración automática de teoremas. La propia lógica de reescritura y Maude ya se han mostrado útiles para desarrollar tales demostradores [Cla00].

Una vez hemos obtenido semánticas operacionales ejecutables, el siguiente paso puede consistir en analizar y demostrar propiedades de tales semánticas, como pueden ser la confluencia o la terminación. Estas propiedades no se refieren a programas concretos escritos en el lenguaje cuya semántica se representa, sino que son metapropiedades aplicables a todos los programas en general. Muchas de ellas se demuestran por inducción estructural sobre las reglas que definen la semántica. A este respecto, nos proponemos estudiar extensiones del demostrador de teoremas ITP [Cla01], desarrollado por Manuel Clavel en el seno del grupo Maude, que permitan la demostración por inducción de tales tipos de propiedades sobre las semánticas ejecutables.

10.2. Protocolos de comunicación

La idea de utilizar Maude como marco semántico ejecutable puede extenderse para dotar de semántica e implementar descripciones que carecían de tal semántica formal. Hemos mostrado cómo se puede utilizar la lógica de reescritura, y Maude en particular,

para especificar y analizar a diferentes niveles de abstracción un protocolo de comunicación como el protocolo de elección de líder del "FireWire", descrito inicialmente en lenguaje natural y código C. También hemos mostrado cómo los aspectos de tiempo del protocolo pueden modelarse de forma sencilla y estructurada en lógica de reescritura, por medio de operaciones que definen el efecto del paso del tiempo y reglas de reescritura que dejan pasar el tiempo. Así mismo, las descripciones pueden analizarse de forma exhaustiva al metanivel, mediante la exploración de todos los estados alcanzables a partir de una configuración inicial, lo que produce una demostración de corrección para una red concreta dada.

Vemos este trabajo como una nueva contribución al área de investigación sobre la especificación y el análisis de varias clases de protocolos de comunicación en Maude, tal y como se describe en [DMT98, DMT00a], así como al desarrollo de la metodología formal resumida en la introducción de esta tesis. En nuestra opinión, es necesario aportar más ejemplos que consoliden la metodología, y desarrollar herramientas que puedan ayudar en la simulación y análisis de dichos ejemplos. A este respecto nos gustaría citar los trabajos realizados por Ölveczky, Meseguer y otros [ÖKM+01, MÖST02].

También hemos estudiado cómo las nuevas facilidades que ofrece la versión Maude 2.0 pueden utilizarse para analizar de forma más sencilla protocolos como el descrito. El reciente desarrollo de un analizador automático de estados para una versión lineal de la lógica temporal [EMS02] ha dado lugar a la posibilidad de aplicar el método de "model checking" a sistemas especificados en un lenguaje como Maude, que es considerablemente más expresivo que los que habitualmente se consideran en estos analizadores. Estudiar su aplicación para la verificación de protocolos de comunicación como el de elección de líder es una línea de trabajo futuro muy interesante.

Anteriormente hemos expresado nuestra pretensión de implementar la semántica del álgebra de procesos E-LOTOS. Esto nos permitiría ejecutar y analizar las descripciones sobre el protocolo de elección de líder que realizamos utilizando E-LOTOS [SV01], que hemos comentado en la Sección 8.8. Hasta el momento no lo hemos podido hacer de una forma mecanizada, ya que no se dispone de implementaciones de E-LOTOS.

10.3. Lenguajes de especificación de la web semántica

La traducción de documentos RDF a módulos orientados a objetos en Maude propuesta y el caso de estudio presentado para ilustrar su uso constituyen un nuevo ejemplo de cómo Maude puede utilizarse como marco semántico ejecutable, en este caso para dar semántica a descripciones puramente sintácticas consiguiendo que el resultado de la traducción pueda integrarse directamente con código ejecutable en el mismo formalismo.

Hemos presentado los primeros resultados de la traducción de RDF y RDFS al lenguaje formal Maude. La traducción, implementada en el propio Maude, produce una versión formal de los datos originales en RDF/RDFS, sin necesidad de añadir información extra en los documentos originales, lo que permite la compatibilidad con otros enfoques. Sin embargo, queda mucho trabajo por hacer en el área de utilización de Maude para la verificación de agentes móviles que utilizan RDF para intercambiar información.

Recientemente se ha extendido el estudio de ontologías para la web, para facilitar el

uso, composición, compartición, etc. de los recursos en la web. Una ontología es una forma de describir el significado y las relaciones entre recursos, con la idea de que, al poner esta información por encima de RDF, se facilite el uso automatizado de esos recursos así como la interoperabilidad semántica entre los sistemas. Entre los diferentes lenguajes que han sido propuestos para la descripción de datos en la web mediante ontologías, destacamos DAML+OIL [Hor02] y OWL [OWL02]. Como trabajo futuro nos gustaría continuar esta línea de trabajo extendiendo la traducción de RDF a Maude para incorporar ontologías a través de lenguajes como los anteriores, que enriquecen considerablemente RDF.

Por otro lado, la traducción de RDF y RDFS a Maude, junto con la integración de Maude en Mobile Maude, permite expresar servicios web mediante agentes móviles en Mobile Maude. Recientemente, el grupo de DAML [DAM] ha propuesto una ontología específica en DAML+OIL, conocida como DAML-S [ABH+02], para describir propiedades y capacidades de servicios web. Este enfoque es declarativo, pues se basa en describir qué hace un servicio y por qué, sin tener que decir cómo. Por tanto, parece razonable tratar de relacionarlo con nuestro enfoque declarativo basado en la lógica de reescritura y Maude. Además, ambos enfoques comparten el interés en la interoperabilidad y composicionalidad, junto con la verificación de propiedades de servicios. Otro de nuestros objetivos futuros en relación con este tema es el desarrollo de semánticas dinámicas para servicios web a través de agentes en Mobile Maude y el estudio de la relación conceptual con DAML-S.

Bibliografía

[ABH+02] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne y K. Sycara. DAML-S: Web service description for the semantic web. En I. Horrocks y J. Hendler, editores, *The Semantic Web - ISWC 2002 First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002. Proceedings*, volumen 2342 de *Lecture Notes in Computer Science*, páginas 348–363. Springer-Verlag, 2002.

[ADV01a] A. Albarrán, F. Durán y A. Vallecillo. From Maude specifications to SOAP distributed implementations: a smooth transition. En *Proceedings VI Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2001, Almagro, Spain*. noviembre 2001.

[ADV01b] A. Albarrán, F. Durán y A. Vallecillo. Maude meets CORBA. En *Proceedings 2nd Argentine Symposium on Software Engineering*. Buenos Aires, Argentina, septiembre 10–11, 2001.

[ADV01c] A. Albarrán, F. Durán y A. Vallecillo. On the smooth implementation of component-based system specifications. En *Proceedings 6th ECOOP International Workshop on Component-Oriented Programming, WCOP'01*. Budapest, Hungary, junio 2001.

[BB87] T. Bolognesi y E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, enero 1987.

[BCM00] J. V. Baalen, J. L. Caldwell y S. Mishra. Specifying and checking fault-tolerant agent-based protocols using Maude. En J. Rash, C. Rouff, W. Truszkowski, D. Gordon y M. Hinchey, editores, *First International Workshop, FAABS 2000, Greenbelt, MD, USA, April 2000. Revised Papers*, volumen 1871 de *Lecture Notes in Artificial Intelligence*, páginas 180–193. Springer-Verlag, 2000.

[BG02] D. Brickley y R. Guha. RDF vocabulary description language 1.0: RDF Schema. W3C Working Draft, 2002. http://www.w3.org/TR/rdf-schema.

[BHL99] T. Bray, D. Hollander y A. Layman. Namespaces in XML, 1999. http://www.w3.org/TR/REC-xml-names/.

[BHMM00] C. Braga, E. H. Haeusler, J. Meseguer y P. D. Mosses. Maude Action Tool: Using reflection to map action semantics to rewriting logic. En T. Rus, editor, *AMAST: 8th International Conference on Algebraic Methodology and Software Technology*, volumen 1816 de *Lecture Notes in Computer Science*, páginas 407–421. Springer-Verlag, 2000.

[BJM00] A. Bouhoula, J.-P. Jouannaud y J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[BLHL01] T. Berners-Lee, J. Hendler y O. Lassila. The semantic web. *Scientific American*, 2001.

[BLMO+02a] M. Bradley, L. Llana, N. Martí-Oliet, T. Robles, J. Salvachua y A. Verdejo. Giving semantics to RDF by mapping into rewriting logic, 2002. Póster presentado en ISWC 2002.

[BLMO+02b] M. Bradley, L. Llana, N. Martí-Oliet, T. Robles, J. Salvachua y A. Verdejo. Transforming information in RDF to rewriting logic. En R. Peña, A. Herranz y J. J. Moreno, editores, *Segundas Jornadas sobre Programación y Lenguajes (PROLE 2002)*, páginas 167–182. 2002.

[BLOMPM96] S. Breitinger, R. Loogen, Y. Ortega-Mallén y R. Peña-Marí. Eden – The paradise of functional concurrent programming. En *Euro-Par'96*, volumen 1123 de *Lecture Notes in Computer Science*, páginas 710–713. Springer-Verlag, 1996.

[BMM98] R. Bruni, J. Meseguer y U. Montanari. Internal strategies in a rewriting implementation of tile systems. En Kirchner y Kirchner [KK98], páginas 95–116. http://www.elsevier.nl/locate/entcs/volume15.html.

[Bra01] C. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. Tesis Doctoral, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brasil, septiembre 2001.

[Bru99] R. Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems*. Tesis Doctoral, Dipartimento di Informatica, Università di Pisa, 1999. Informe técnico TD-1/99. http://www.di.unipi.it/phd/tesi/tesi_1999/TD-1-99.ps.gz.

[BS01] J. Bryans y C. Shankland. Implementing a modal logic over data and processes using XTL. En Kim et al. [KCKL01], páginas 201–218.

[BT80] J. Bergstra y J. Tucker. Characterization of computable data types by means of a finite equational specification method. En J. W. de Bakker y J. van Leeuwen, editores, *Automata, Languages and Programming, Seventh Colloquium, Noordwijkerhout, The Netherlands*, volumen 81 de *Lecture Notes in Computer Science*, páginas 76–90. Springer-Verlag, 1980.

[Car99] L. Cardelli. Abstractions for mobile computations. En J. Vitek y C. Jensen, editores, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volumen 1603 de *Lecture Notes in Computer Science*, páginas 51–94. Springer-Verlag, 1999.

[CDE+98a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet y J. Meseguer. Metalevel computation in Maude. En Kirchner y Kirchner [KK98], páginas 3–24. http://www.elsevier.nl/locate/entcs/volume15.html.

[CDE+98b] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Maude as a metalanguage. En Kirchner y Kirchner [KK98], páginas 237–250. http://www.elsevier.nl/locate/entcs/volume15.html.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, enero 1999. http://maude.cs.uiuc.edu/manual.

[CDE+00a] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. Quesada. *A Maude Tutorial*. Computer Science Laboratory, SRI International, marzo 2000. http://maude.cs.uiuc.edu/tutorial.

[CDE+00b] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Towards Maude 2.0. En Futatsugi [Fut00], páginas 297–318. http://www.elsevier.nl/locate/entcs/volume36.html.

[CDE+00c] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Using Maude. En T. Maibaum, editor, *Fundamental Approaches to Software Engineering, Third International Conference, FASE 2000, Held as Part of ETAPS 2000, Berlin, Germany, March/April 2000, Proceedings*, volumen 1783 de *Lecture Notes in Computer Science*, páginas 371–374. Springer-Verlag, 2000.

[CDE+01] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet y J. Meseguer. Language prototyping in the Maude metalanguage. En F. Orejas, F. Cuartero y D. Cazorla, editores, *Actas PROLE 2001, Primeras Jornadas sobre Programación y Lenguajes, Almagro (Ciudad Real), 23-24 Noviembre 2001*, páginas 93–110. Universidad de Castilla La Mancha, 2001.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[Cla98] M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*. Tesis Doctoral, Universidad de Navarra, 1998.

[Cla00] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[Cla01] M. Clavel. The ITP tool. En A. Nepomuceno, J. F. Quesada y J. Salguero, editores, *Logic, Language and Information. Proceedings of the First Workshop on Logic and Language*, páginas 55–62. Kronos, 2001.

[CM96] M. Clavel y J. Meseguer. Reflection and strategies in rewriting logic. En Meseguer [Mes96a], páginas 125–147. http://www.elsevier.nl/locate/entcs/volume4.html.

[CMS95] R. Cleaveland, E. Madelaine y S. T. Sims. A front-end generator for verification tools. En *Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volumen 1019 de *Lecture Notes in Computer Science*, páginas 153–173. Springer-Verlag, 1995.

[CMS01] M. Calder, S. Maharaj y C. Shankland. An adequate logic for Full LOTOS. En J. Oliveira y P. Zave, editores, *FME 2001: Formal Methods for Increasing Software Productivity*, volumen 2021 de *Lecture Notes in Computer Science*, páginas 384–395. Springer-Verlag, 2001.

[CMS02] M. Calder, S. Maharaj y C. Shankland. A modal logic for Full LOTOS based on symbolic transition systems. *The Computer Journal*, 45(1):55–61, 2002.

[Coq88] T. Coquand. The calculus of constructions. *Information and Control*, 76:95–120, 1988.

[CS00] M. Calder y C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. Informe técnico CSM-159, University of Stirling, 2000.

[CS01] M. Calder y C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. En Kim et al. [KCKL01], páginas 184–200.

[CS02] R. Cleaveland y S. T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, enero 2002.

[DAM] The DARPA Agent Markup Language. http://www.daml.org/.

[DELM00] F. Durán, S. Eker, P. Lincoln y J. Meseguer. Principles of Mobile Maude. En D. Kotz y F. Mattern, editores, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volumen 1882 de *Lecture Notes in Computer Science*. Springer-Verlag, septiembre 2000.

[Des84] T. Despeyroux. Executable specification of static semantics. En G. Kahn, D. B. MacQueen y G. D. Plotkin, editores, *Semantics of Data Types*, volumen 173 de *Lecture Notes in Computer Science*, páginas 215–233. Springer-Verlag, 1984.

[Des88] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Research Report 94, INRIA, 1988.

[DGRV97] M. Devillers, W. Griffioen, J. Romijn y F. Vaandrager. Verification of a leader election protocol — formal methods applied to IEEE 1394. Informe técnico CSI-R9728, Computing Science Institute, University of Nijmegen, 1997.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DM00a] G. Denker y J. Millen. CAPSL integrated protocol environment. En D. Maughan, G. Koob y S. Saydjari, editores, *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25-27, 2000*, páginas 207–222. IEEE Computer Society Press, 2000. http://schafercorp-ballston.com/discex/.

[DM00b] F. Durán y J. Meseguer. A Church-Rosser checker tool for Maude equational specifications, 2000. Manuscrito, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers.

[DMT98] G. Denker, J. Meseguer y C. L. Talcott. Protocol specification and analysis in Maude. En N. Heintze y J. Wing, editores, *Proceedings of Workshop on Formal Methods and Security Protocols, June 25, 1998, Indianapolis, Indiana*. 1998. http://www.cs.bell-labs.com/who/nch/fmsp/index.html.

[DMT00a] G. Denker, J. Meseguer y C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. En D. Maughan, G. Koob y S. Saydjari, editores, *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25–27, 2000*, páginas 251–265. IEEE Computer Society Press, 2000. http://schafercorp-ballston.com/discex/.

[DMT00b] G. Denker, J. Meseguer y C. L. Talcott. Rewriting semantics of meta-objects and composable distributed services. En Futatsugi [Fut00], páginas 407–427. http://www.elsevier.nl/locate/entcs/volume36.html.

[Dur99] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. Tesis Doctoral, Universidad de Málaga, junio 1999. http://maude.cs.uiuc.edu/papers.

[Dur00a] F. Durán. Coherence checker and completion tools for Maude specifications, 2000. Manuscrito, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers.

[Dur00b] F. Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications, 2000. Manuscrito, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers.

[DV01] F. Durán y A. Vallecillo. Writing ODP enterprise specifications in Maude. En *Proceedings Workshop On Open Distributed Processing: Enterprise, Computation, Knowledge, Engineering and Realisation, WOODPECKER 2001*. Setúbal, Portugal, julio 2001. http://www.lcc.uma.es/~av/Publicaciones/01/ITI-2001-8.pdf.

[DV02] F. Durán y A. Verdejo. A conference reviewing system in Mobile Maude. En Gadducci y Montanari [GM02], páginas 79–95. http://www.elsevier.nl/locate/entcs/volume71.html.

[EM85] H. Ehrig y B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[EMS02] S. Eker, J. Meseguer y A. Sridharanarayanan. The Maude LTL model checker. En Gadducci y Montanari [GM02], páginas 115–141. http://www.elsevier.nl/locate/entcs/volume71.html.

[FGH+88] A. Felty, E. Gunter, J. Hannan, D. Miller, G. Nadathur y A. Scedrov. Lambda Prolog: An extended logic programming language. En E. Lusk y R. Overbeek, editores, *Proceedings on the 9th International Conference on Automated Deduction*, volumen 310 de *Lecture Notes in Computer Science*, páginas 754–755. Springer-Verlag, 1988.

[FHS01] M. M. Furio Honsell y I. Scagnetto. π-calculus in (co)inductive-type theory. *Theoretical Computer Science*, 253(2):239–285, 2001.

[FS02] C. Fidge y C. Shankland. But what if I don't want to wait forever? *Formal Aspects of Computing*, 14(3), 2002.

[FT00] J. L. Fernández y A. Toval. Can intuition become rigorous? Foundations for UML model verification tools. En F. M. Titsworth, editor, *International Symposium on Software Reliability Engineering*, páginas 344–355. IEEE Press, San José, California, octubre 2000.

[FT01] J. L. Fernández y A. Toval. Seamless formalizing the UML semantics through metamodels. En K. Siau y T. Halpin, editores, *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, páginas 224–248. Idea Group Publishing, 2001.

[Fut00] K. Futatsugi, editor. *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volumen 36 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume36.html.

[GHW85] J. V. Guttag, J. J. Horning y J. M. Wing. Larch in five easy pieces. Informe técnico 5, Digital Equipment Corporation, Systems Research Centre, 1985.

[GM93] M. Gordon y T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GM02] F. Gadducci y U. Montanari, editores. *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volumen 71 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. http://www.elsevier.nl/locate/entcs/volume71.html.

[Hay02] P. Hayes. RDF model theory. W3C Working Draft, 2002. http://www.w3.org/TR/rdf-mt.

[Hen90] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Willey & Sons, 1990.

[HHOM02] M. Hidalgo-Herrero y Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.

[Hir97] D. Hirschkoff. A full formalisation of π-calculus theory in the calculus of constructions. En *Proc. 10th International Theorem Proving in Higher Order Logic Conference*, volumen 1275 de *Lecture Notes in Computer Science*, páginas 153–169. Springer-Verlag, 1997.

[HKPM02] G. Huet, G. Kahn y C. Paulin-Mohring. The Coq proof assistant: a tutorial. version 7.2. Informe técnico 256, INRIA, 2002.

[HL95] M. Hennessy y H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[HLDQ+01] G. Huecas, L. Llana-Díaz, J. Quemada, T. Robles y A. Verdejo. Process calculi: E-LOTOS. En H. Bowman y J. Derrick, editores, *Formal Methods for Distributed Processing. A Survey of Object-Oriented Aproaches*, capítulo 5, páginas 77–104. Cambridge University Press, 2001.

[HLDRV99] G. Huecas, L. Llana-Díaz, T. Robles y A. Verdejo. E-LOTOS: An overview. En Marsan et al. [MQRS99], páginas 94–102.

[HM85] M. Hennessy y R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, enero 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hor02] I. Horrocks. DAML+OIL: A description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25(1):4–9, 2002.

[IEE95] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High Performance Serial Bus. Std 1394-1995*, agosto 1995.

[IMW+97] H. Ishikawa, J. Meseguer, T. Watanabe, K. Futatsugi y H. Nakashima. On the semantics of GAEA — An object-oriented specification of a concurrent reflective language in rewriting logic. En *Proceedings IMSA'97*, páginas 70–109. Information-Technology Promotion Agency, Japan, 1997.

[ISO89] ISO/IEC. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*. International Standard 8807, International Organization for standardization — Information Processing Systems — Open Systems Interconnection, Geneva, septiembre 1989.

[ISO01] ISO/IEC. Information technology — Enhancements to LOTOS (E-LOTOS). International Standard ISO/IEC FDIS 15437, 2001.

[JHA+96] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu y M. Sighireanu. CADP: a protocol validation and verification toolbox. En R. Alur y T. A. Henzinger, editores, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volumen 1102 de *Lecture Notes in Computer Science*, páginas 437–440. Springer-Verlag, 1996.

[Kah87] G. Kahn. Natural semantics. Informe técnico 601, INRIA Sophia Antipolis, febrero 1987.

[KCKL01] M. Kim, B. Chin, S. Kang y D. Lee, editores. *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems*. Kluwer Academic Publishers, 2001.

[KK98] C. Kirchner y H. Kirchner, editores. *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volumen 15 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.

[Lam69] J. Lambek. Deductive systems and categories II. En *Category Theory, Homology Theory and their Applications I*, volumen 86 de *Lecture Notes in Mathematics*, páginas 76–122. Springer-Verlag, 1969.

[LMOM94] P. Lincoln, N. Martí-Oliet y J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. En G. E. Blelloch, K. M. Chandy y S. Jagannathan, editores, *Specification of Parallel Algorithms, DIMACS Workshop, May 9-11, 1994*, páginas 309–339. American Mathematical Society, 1994.

[LP92] Z. Luo y R. Pollack. The LEGO proof development system: A user's manual. Informe técnico ECS-LFCS-92-211, University of Edinburgh, 1992.

[LS99] O. Lassila y R. R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation, 22 February, 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[Mel94] T. F. Melham. A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing*, 1(1):50–76, 1994.

[Mes90] J. Meseguer. Rewriting as a unified model of concurrency. En J. C. M. Baeten y J. W. Klop, editores, *CONCUR'90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 1990, Proceedings*, volumen 458 de *Lecture Notes in Computer Science*, páginas 384–400. Springer-Verlag, 1990.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. En G. Agha, P. Wegner y A. Yonezawa, editores, *Research Directions in Concurrent Object-Oriented Programming*, páginas 314–390. The MIT Press, 1993.

[Mes96a] J. Meseguer, editor. *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volumen 4 de *Electronic Notes in Theoretical Computer Science*. Elsevier, septiembre 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[Mes96b] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. En U. Montanari y V. Sassone, editores, *CONCUR'96: Concurrency Theory,*

7th International Conference, Pisa, Italy, August 26–29, 1996, Proceedings, volumen 1119 de *Lecture Notes in Computer Science*, páginas 331–372. Springer-Verlag, 1996.

[Mes98] J. Meseguer. Research directions in rewriting logic. En U. Berger y H. Schwichtenberg, editores, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, páginas 347–398. Springer-Verlag, 1998.

[MFW92] J. Meseguer, K. Futatsugi y T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. En *Proceedings of the 1992 International Symposium on New Models for Software Architecture*, páginas 61–106. Research Institute of Software Engineering, Tokyo, Japan, noviembre 1992.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MOM93] N. Martí-Oliet y J. Meseguer. Rewriting logic as a logical and semantic framework. Informe técnico SRI-CSL-93-05, SRI International, Computer Science Laboratory, agosto 1993. Aparecere también en D. M. Gabbay y F. Guenthner, editores, *Handbook of Philosophical Logic, Second Edition, Volume 9*, paginas 1–87. Kluwer Academic Publishers, 2002. http://maude.cs.uiuc.edu/papers.

[MOM99] N. Martí-Oliet y J. Meseguer. Action and change in rewriting logic. En R. Pareschi y B. Fronhöfer, editores, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volumen 12 de *Applied Logic Series*, páginas 1–53. Kluwer Academic Publishers, 1999.

[MOM02] N. Martí-Oliet y J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[Mos99] P. Mosses. Foundations of modular SOS. En M. Kutylowski, L. Pacholksi y T. Wierzbicki, editores, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99 Szklarska Poreba, Poland, September 6–10, 1999, Proceedings*, volumen 1672 de *Lecture Notes in Computer Science*, páginas 70–80. Springer-Verlag, 1999. Una versión extendida aparece como informe técnico RS-99-54, BRICS, Dept. of Computer Science, University of Aarhus.

[MÖST02] J. Meseguer, P. Ölveczky, M.-O. Stehr y C. L. Talcott. Maude as a wide-spectrum framework for formal modeling and analysis of active networks. En DARPA Active Networks Conference and Exposition (DANCE), San Francisco, 2002.

[MQRS99] M. A. Marsan, J. Quemada, T. Robles y M. Silva, editores. *Formal Methods and Telecommunications (FM&T'99)*. Prensas Universitarias de Zaragoza, septiembre 1999.

[MRS02] S. Maharaj, J. Romijn y C. Shankland. IEEE 1394 tree identify protocol: Introduction to the case study. *Formal Aspects of Computing*, 14(3), 2002.

[MS00] S. Maharaj y C. Shankland. A Survey of Formal Methods Applied to Leader Election in IEEE 1394. *Journal of Universal Computer Science*, 6(11):1145–1163, noviembre 2000.

[Nes96] M. Nesi. Mechanising a modal logic for value-passing agents in HOL. En B. Steffen y D. Caucal, editores, *Infinity'96, First International Workshop on Verification of Infinite State Systems*, volumen 5 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[Nes99] M. Nesi. Formalising a value-passing calculus in HOL. *Formal Aspects of Computing*, 11:160–199, 1999.

[Nip98] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[NPW02] T. Nipkow, L. C. Paulson y M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volumen 2283 de *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[ÖKM+01] P. C. Ölveczky, M. Keaton, J. Meseguer, C. L. Talcott y S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. En H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Held as Part of ETAPS 2001, Genova, Italy, April 2001, Proceedings*, volumen 2029 de *Lecture Notes in Computer Science*, páginas 333–347. Springer-Verlag, 2001. http://maude.cs.uiuc.edu/papers.

[Ölv00] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. Tesis Doctoral, University of Bergen, Norway, 2000. http://maude.cs.uiuc.edu/papers.

[ÖM02] P. C. Ölveczky y J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[OWL02] OWL web ontology language 1.0 reference, 2002. http://www.w3.org/TR/owl-ref/.

[Pet94] M. Pettersson. RML — A new language and implementation for natural semantics. En M. Hermenegildo y J. Penjam, editores, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94*, volumen 844 de *Lecture Notes in Computer Science*, páginas 117–131. Springer-Verlag, 1994.

[Pet96] M. Pettersson. A compiler for natural semantics. En T. Gyimothy, editor, *Compiler Construction, 6th International Conference*, volumen 1060 de *Lecture Notes in Computer Science*, páginas 177–191. Springer-Verlag, 1996.

[Pit03] I. Pita. *Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura*. Tesis Doctoral, Facultad de Matemáticas, Universidad Complutense de Madrid, 2003.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Informe técnico DAIMI FN-19, Computer Science Department, Aarhus University, septiembre 1981.

[PM94] C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. En *Proc. TLCA'94*, volumen 664 de *Lecture Notes in Computer Science*, páginas 328–345. Springer-Verlag, 1994.

[RC02] B. Ross y M. Calder. Computing symbolic bisimulations for Full LOTOS. *Formal Aspects of Computing*, 2002. Pendiente de aceptación para ser publicado.

[Rey98] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[RHB01] C. Röeckl, D. Hirschkoff y S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the π-calculus and mechanizing the theory of contexts. En F. Honsell y M. Miculan, editores, *Proc. FOSSACS'01*, volumen 2030 de *Lecture Notes in Computer Science*, páginas 364–378. Springer-Verlag, 2001.

[Röe01a] C. Röeckl. A first-order syntax for the π-calculus in Isabelle/HOL using permutations. En S. Ambler, R. Crole y A. Momigliano, editores, *Proc. MERLIN'01*, volumen 58.1 de *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[Röe01b] C. Röeckl. *On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems*. Tesis Doctoral, Fakultät für Informatik, Technische Universität München, 2001.

[Rom01] J. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, septiembre 2001. Volumen especial sobre el Fourth International Workshop on Formal Methods for Industrial Critical Systems.

[SBM+02] C. Shankland, J. Bryans, S. Maharaj, M. Calder, B. Ross, L. Morel, P. Robinson y A. Verdejo. A symbolic approach to infinite state systems. *International Journal of Foundations of Computer Science*, 2002. Pendiente de aceptación para ser publicado.

[SC02] C. Shankland y M. Calder. Developing Implementation and Extending Theory: A symbolic approach about LOTOS. EPSRC project CR/M07779/01, 2002. http://www.cs.stir.ac.uk/diet.

[SM98] M. Sighireanu y R. Mateescu. Verification of the link layer protocol of the IEEE-1394 serial bus ("FireWire"): an experiment with E-LOTOS. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):68–88, 1998.

[SM99] M.-O. Stehr y J. Meseguer. Pure type systems in rewriting logic. En *Proceedings of LFM'99: Workshop on Logical Frameworks and Meta-languages*. Paris, France, septiembre 1999. http://www.cs.bell-labs.com/~felty/LFM99/.

[SMR01] C. Shankland, S. Maharaj y J. Romijn. International workshop on application of formal methods to IEEE 1394 standard, marzo 2001. http://www.cs.stir.ac.uk/firewire-workshop.

[Spr98] C. Sprenger. A verified model checker for the modal μ-calculus in Coq. En B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volumen 1384 de *Lecture Notes in Computer Science*, páginas 167–183. Springer-Verlag, 1998.

[ST02] M.-O. Stehr y C. L. Talcott. PLAN in Maude: Specifying an active network programming language. En Gadducci y Montanari [GM02], páginas 195–215. http://www.elsevier.nl/locate/entcs/volume71.html.

[Ste00] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. En Futatsugi [Fut00], páginas 71–92. http://www.elsevier.nl/locate/entcs/volume36.html.

[Sti96] C. Stirling. Modal and temporal logics for processes. En F. Moller y G. Birtwistle, editores, *Logics for Concurrency: Structure vs Automata*, volumen 1043 de *Lecture Notes in Computer Science*, páginas 149–237. Springer-Verlag, 1996.

[SV99a] C. Shankland y A. Verdejo. Time, E-LOTOS, and the FireWire. En Marsan et al. [MQRS99], páginas 103–119.

[SV99b] M. I. A. Stoelinga y F. W. Vaandrager. Root contention in IEEE 1394. En J.-P. Katoen, editor, *Proceedings 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, volumen 1601 de *Lecture Notes in Computer Science*, páginas 53–74. Springer-Verlag, 1999.

[SV01] C. Shankland y A. Verdejo. A case study in abstraction using E-LOTOS and the FireWire. *Computer Networks*, 37(3–4):481–502, 2001.

[SZ98] C. Shankland y M. van der Zwaag. The Tree Identify Protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10:509–531, 1998.

[Ter95] D. Terrasse. Encoding natural semantics in Coq. En V. S. Alagar, editor, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology*, volumen 936 de *Lecture Notes in Computer Science*, páginas 230–244. Springer-Verlag, 1995.

[TF00] A. Toval y J. L. Fernández. Formally modeling UML and its evolution: A holistic approach. En S. F. Smith y C. L. Talcott, editores, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, 2000, Stanford, California, USA*, páginas 183–206. Kluwer Academic Publishers, 2000.

[TSMO02] P. Thati, K. Sen y N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. En Gadducci y Montanari [GM02], páginas 217–237. http://www.elsevier.nl/locate/entcs/volume71.html.

[Tur92] K. Turner. *Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL.* John Wiley and Sons Ltd., 1992.

[URI00] IETF Uniform Resource Identifiers (URI) Working Group, 2000. http://ftp.ics.uci.edu/pub/ietf/uri/.

[Ver02a] A. Verdejo. LOTOS symbolic semantics in Maude. Informe técnico 122-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, enero 2002. http://dalila.sip.ucm.es/~alberto.

[Ver02b] A. Verdejo. A tool for Full LOTOS in Maude. Informe técnico 123-02, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, abril 2002. http://dalila.sip.ucm.es/~alberto.

[Ver02c] A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. En D. Peled y M. Vardi, editores, *Formal Techniques for Networked and Distributed Systems — FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 2002 Proceedings*, volumen 2529 de *Lecture Notes in Computer Science*, páginas 292–307. Springer-Verlag, 2002.

[Vir96] P. Viry. Input/output for ELAN. En Meseguer [Mes96a], páginas 51–64. http://www.elsevier.nl/locate/entcs/volume4.html.

[Vit94] M. Vittek. *ELAN: Un Cadre Logique pour le Prototypage de Langages de Programmation avec Contraintes*. Tesis Doctoral, Université Henri Poincaré – Nancy I, noviembre 1994.

[VMO00a] A. Verdejo y N. Martí-Oliet. Executing and verifying CCS in Maude. Informe técnico 99-00, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, febrero 2000.

[VMO00b] A. Verdejo y N. Martí-Oliet. Executing E-LOTOS processes in Maude. En H. Ehrig, M. Grosse-Rhode y F. Orejas, editores, *INT 2000, Integration of Specification Techniques with Applications in Engineering, Extended Abstracts Technical report 2000/04, Technische Universitat Berlin, March 2000*, páginas 49–53. 2000.

[VMO00c] A. Verdejo y N. Martí-Oliet. Implementing CCS in Maude. En T. Bolognesi y D. Latella, editores, *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) October 10–13, 2000, Pisa, Italy*, páginas 351–366. Kluwer Academic Publishers, 2000.

[VMO02a] A. Verdejo y N. Martí-Oliet. Executing and verifying CCS in Maude. *Formal Methods and System Design*, 2002. Enviado para su publicación, pendiente de aceptación.

[VMO02b] A. Verdejo y N. Martí-Oliet. Implementing CCS in Maude 2. En Gadducci y Montanari [GM02], páginas 239–257. http://www.elsevier.nl/locate/entcs/volume71.html.

[VPMO00] A. Verdejo, I. Pita y N. Martí-Oliet. The leader election protocol of IEEE 1394 in Maude. En Futatsugi [Fut00], páginas 385–406. http://www.elsevier.nl/locate/entcs/volume36.html.

[VPMO01a] A. Verdejo, I. Pita y N. Martí-Oliet. The leader election protocol of IEEE 1394 in Maude. Informe técnico 118-01, Dpto. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001. http://dalila.sip.ucm.es/~alberto.

[VPMO01b] A. Verdejo, I. Pita y N. Martí-Oliet. Specification and verification of the leader election protocol of IEEE 1394 in rewriting logic. En S. Maharaj, J. Romijn y C. Shankland, editores, *Proceedings International Workshop on Application of Formal Methods to IEEE 1394 Standard*, páginas 39–43. Berlin, marzo 2001.

[VPMO02] A. Verdejo, I. Pita y N. Martí-Oliet. Specification and verification of the tree identify protocol of IEEE 1394 in rewriting logic. *Formal Aspects of Computing*, 14(3), 2002.

[W3C] Resource Description Framework (RDF) / W3C Semantic Web Activity. http://www.w3.org/RDF/.

[WMG00] B.-Y. Wang, J. Meseguer y C. A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. En P.-A. Hsiung, editor, *Proceedings International Workshop on Distributed System Validation and Verification, Taipei, Taiwan*, páginas 49–56. 2000.

[XML] Extensible markup language (XML). http://www.w3.org/XML/.

[XML01] XML Schema part 0: Primer. W3C Recommendation, 2001. http://www.w3.org/TR/xmlschema-0/.

[YL97] S. Yu y Z. Luo. Implementing a model checker for LEGO. En J. Fitzgerald, C. B. Jones y P. Lucas, editores, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volumen 1313 de *Lecture Notes in Computer Science*, páginas 442–458. Springer-Verlag, 1997.