UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

SISTEMAS DE REDUCCION COMPUESTOS. EL CASO DE GAMMA.

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

EDGAR JOSUE BERMUDEZ CONTRERAS

DR. FRANCISCO HERNANDEZ QUIROZ

FACULTAD DE CIENCIAS
SECCION ESCOLAR

UNIVERSIDAD NACIONAL
AVENOMA DE
MEXICO

M. EN C. ELENA DE OTEYZA DE OTEYZA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

"Sistemas de Reducción Compuestos. El caso de Gamma"

realizado por BERMUDEZ CONTRERAS EDGAR JOSUE

con número de cuenta 09757212-1 , quién cubrió los créditos de la carrera de C. DE LA COMPUTACION

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	DR. FRANCISCO HERNANDEZ QUIROZ	
Propietario	DR. MANUEL ROMERO SALCEDO	
Propietario	MTRO. MIGUEL CARRILLO BARAJAS	
Suplente	M. EN I. MARIA DE LA LUZ GASCA SOTO	
Suplente	DRA. HANNA OKTABA	

Consejo Departamental de Matemáticas

DRA. AMPARO LOPEZ GAONA

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

Gracias

A Dios

A mis padres y hermanos

A TODOS mis amigos

A mis maestros

Índice general

Capítulo 1

Introducción

1.1. Los lenguajes de programación

Debido a la importancia que los lenguajes de programación tienen dentro del desarrollo de software y el papel que las aplicaciones tienen en nuestra vida diaria, es indispensable que los programas sean eficientes y confiables. Debido a esto, surge la necesidad de analizar los lenguajes de programación con el fin de verificar que los programas sean correctos, es decir, que cumplan con las tareas para los que fueron hechos.

La forma en que podemos saber cómo se comporta un programa con el fin de verificar que sea correcto es teniendo una manera de entender el significado de un programa, a esto se le llama la semántica de un lenguaje. Se busca entonces, una herramienta que permita hacer demostraciones formales sobre propiedades que reflejen el comportamiento que nosotros esperamos del programa. Por esta razón, es necesario estudiar a profundidad la semántica del lenguaje en cuestión.

En la actualidad hay una gran cantidad de lenguajes de programación. Algunos presentan ciertas ventajas sobre otros dependiendo del problema, las características que se buscan o los recursos que se desean optimizar. En base a esto y algunas otras cosas, un programador puede decidir en qué lenguaje desea hacer el programa o sistema que resuelva el problema planteado.

Últimamente ha tomado gran auge la programación paralela. Con ella se busca optimizar el desempeño, ya que se pueden realizar tareas al mismo tiempo, prescindiendo de la secuencialidad de estas últimas.

En muchos casos esta forma de resolver los problemas resulta ser mucho más eficiente que la manera convencional.

El presente trabajo tiene como punto central de estudio, la semántica de un lenguaje de naturaleza paralela: Gamma.

A través de este trabajo buscamos encontrar formas de demostración de programas escritos en Gamma proponiendo una transformación para los programas en sistemas de reducción compuestos. Esta abstracción, junto con una semántica y lógica propuestas, permitirán tener un sistema formal de demostración y de esta forma, la verificación de los programas de la que ya hemos hablado.

1.2. Gamma y los SRC

El lenguaje de programación Gamma está basado en la metáfora de una reacción química. Lo que se busca es simular la forma en que reaccionan las moléculas dentro de una solución. En un programa de Gamma, tenemos una única estructura: el multiconjunto. Los multiconjuntos son simplemente conjuntos donde se permite la repetición de los elementos. Esta estructura es lo que simula la solución y los elementos del multiconjunto simulan las moléculas. Como sabemos, para que algunas moléculas en la solución reaccionen entre sí, deben cumplirse ciertas condiciones, cuando estas condiciones se dejan de cumplir decimos que se alcanza un equilibrio y las moléculas no reaccionan más. Lo mismo pasa en el multiconjunto, se tienen ciertas condiciones sobre los elementos del multiconjunto que se deben cumplir para que se lleven a cabo acciones sobre éstos y de esta forma, que se transformen en otros elementos del multiconjunto. Una vez que ya no existen elementos en el multiconjunto que cumplan con la condición dada, el multiconjunto se queda estable y decimos que esa reacción ha concluido.

Si consideramos ahora que varias reacciones pueden llevarse a cabo en un multiconjunto, es decir, hay varias condiciones y varias acciones sobre los elementos que cumplen estas condiciones, es fácil pensar en que se pueden estar ejecutando varios programas de Gamma al mismo tiempo. Nos damos cuenta que de esta forma se tiene de manera natural el paralelismo dentro del lenguaje pues los componentes de la solución

reaccionan unos con otros apareciendo de forma natural la composición paralela dentro del lenguaje.

En este trabajo estudiamos a fondo la semántica de Gamma con el fin de demostrar propiedades de sus programas, así como de buscar una generalización del lenguaje con un aparato matemático que nos permita tener herramientas mejor estudiadas y propiedades mejor conocidas: los Sistemas de Reducción Compuestos (SRC). Es decir, tenemos la tarea de transformar los programas de Gamma en un tipo especial de Sistemas de Reducción Compuestos como se verá más adelante.

Además de trabajar con Gamma, también se estudia una variante de este lenguaje, Gamma estructurado. Esta variante surge por la necesidad de tener mayor control sobre la estructura de Gamma: el multiconjunto. En Gamma estructurado se introducen algunas restricciones sobre los multiconjuntos, lo que nos da los multiconjuntos estructurados. En los multiconjuntos estructurados se tienen direcciones que satisfacen ciertas relaciones y que están asociadas a valores de los elementos.

Ahora que tenemos una idea más clara del objeto de estudio de este trabajo veamos cómo se llevarán a cabo las tareas antes mencionadas.

1.3. Un panorama general

El presente trabajo está conformado por siete capítulos. En los capítulos 2 y 3 nos situamos en el campo donde estamos trabajando.

En el capítulo 2 se busca la comprensión de la importancia y la utilidad que tienen los lenguajes de programación, así como la forma en que éstos han venido cambiando.

El capítulo 3 nos da un panorama amplio de las herramientas con las que contamos para realizar la tarea que emprende esta tesis mostrando diversas formas de estudiar la semántica de los lenguajes (explicada de manera muy breve anteriormente) y las ventajas y aportaciones que tiene cada una de las semánticas presentadas a través del estudio de un lenguaje muy sencillo. Con el estudio de la semántica de los lenguajes de programación que se hace en este capítulo, podremos abordar el

tema de los sistemas de reducción compuestos[1] (SRC), así como el de la transformación de programas en capítulos posteriores. Al final de este capítulo se muestra una aproximación al paralelismo, se explica en qué consiste y el por qué esta forma de cómputo ha sido importante para las ciencias computacionales. El estudio del paralelismo se hace mediante una extensión del sencillo lenguaje que veníamos estudiando. De esta forma se deja en claro las cuestiones que se busca abordar a través del estudio de los SRC.

En el capítulo 4 se presentan los SRC, se estudia la semántica operacional para estos sistemas con el fin de entender de manera más profunda su comportamiento y se muestran algunos ejemplos de estos sistemas. Una vez que se ha estudiado mejor a los SRC, se introducen algunos "objetos matemáticos" (trazas y algunas operaciones sobre ellas) con el fin de construir una lógica sobre los términos de los SRC que permita hacer demostraciones sobre las propiedades de los términos como se verá en el capítulo donde se aplican las transformaciones propuestas en este trabajo a un ejemplo concreto.

En el capítulo 5 se aborda el tema de Gamma, el lenguaje que usa la metáfora de la reacción química para simular el paralelismo en el lenguaje de programación de manera natural. Se revisan algunas cuestiones del lenguaje como sintaxis, semántica y se plantea también una variante del lenguaje: Gamma estructurado, que surge por la necesidad de tener mayor control sobre la estructura del lenguaje: el multiconjunto. También se estudia la semántica de esta variante de Gamma y los tipos estructurados.

En el capítulo 6 se tiene una generalización del formalismo de Gamma, buscando ver a los programas de Gamma y Gamma estructurados como un objeto matemático más abstracto: los SRC. Con esto se puede saber si Gamma y Gamma estructurado pertenecen a una clase de lenguajes paralelos que ha sido ampliamente estudiada y de esta forma, conocer propiedades de estos dos lenguajes. Algunas de estas propiedades nos permitirán hacer demostraciones sobre la corrección en los programas de estos lenguajes como se estudia en el capítulo siguiente.

En el capítulo 7 se presenta una aplicación donde se ven reflejados

[1] de aquí en adelante también los denotaremos simplemente como SRC

los resultados de este trabajo, se presenta una aplicación con memorias virtuales compartidas (MVC). La aplicación consiste en la presentación de un protocolo de coherencia para un sistema de MVC. Mediante esta aplicación se emplean la transformación propuesta para generalizar los programas de Gamma (ya que el protocolo para las MVC está escrito en este lenguaje), así como la demostración de propiedades usando la lógica propuesta en el capítulo de los SRC. Las propiedades que se demuestran forman un invariante, de tal forma que se puede demostrar que el programa en Gamma es correcto.

Por último se presenta un breve capítulo con las conclusiones del trabajo y posibles rutas a futuro.

1.4. Las bases de la investigación

Este trabajo es parte de la investigación del proyecto "Semántica del paralelismo: en busca de una unificación" que fue financiado por la Red de Desarrollo e Investigación en Informática (REDII) y a cargo del Dr. Francisco Hernández Quiroz.

Cabe señalar que durante seis meses se recibió un apoyo económico para el desarrollo de este trabajo en el tiempo de investigación. Este apoyo fue recibido dentro del Programa de Becas para la Elaboración de Tesis de Licenciatura en Proyectos de Investigación (Probetel) de la UNAM.

La hipótesis de partida para este proyecto fue que la generalización del modelo semántico de Gamma pudiera ser la base para un modelo semántico de los sistemas de reducción compuestos y, de este modo, de una subclase de lenguajes paralelos. Finalmente, se buscó que el modelo semántico permitiera desarrollar algunas técnicas de verificación y transformación de programas.

Los objetivos del proyecto fueron los siguientes:

1. Extraer un modelo matemático más abstracto de la semántica de Gamma diseñada por [Hernández, 1999].

2. Caracterizar qué lenguajes de programación paralela pueden considerarse sistemas compuestos de reducción.

3. Formular un modelo semántico para todos los sistemas de reducción compuestos basado en la versión abstracta de la semántica de Gamma.

4. Determinar si Gamma de orden superior es un sistema de reducción compuesto.

5. En caso de que no lo sea, corresponde investigar de qué forma se puede extender nuestro modelo semántico para cubrir este lenguaje.

6. Del mismo modo que en [Hernández, 1999], se utilizará la semántica para desarrollar técnicas de verificación y transformación de programas paralelos.

Conforme a los objetivos mencionados, surgieron dos trabajos de tesis de este proyecto. Dichos trabajos tienen las mismas bases de investigación: parten de estudiar la semántica de los lenguajes y las estructuras básicas para entender Gamma. Después de este punto cada tesis toma caminos diferentes. La presente trata de abocarse a los objetivos primero, segundo y tercero. Mientras que el trabajo de tesis de Gustavo de La Cruz Martínez [De La Cruz Martínez, 2002] se enfoca a los objetivos cuarto y quinto.

Empecemos entonces con el estudio de los lenguajes de programación para comprender más adelante la forma en que podemos describir el comportamiento de los programas.

Capítulo 2

Los lenguajes de programación

Partiendo de la idea de que los lenguajes de programación nos sirven para describir algoritmos, nos podemos dar cuenta de la importancia que éstos tienen en el mundo de la computación.

2.1. Su importancia

Los lenguajes de programación son empleados en cualquier aplicación de software que podamos pensar, para desarrollar cualquier programa, por más sencillo que sea, debemos utilizar un medio para decirle a la computadora lo que queremos que haga. Por esta razón los lenguajes de programación son parte del corazón de la ciencia de la computación.

Sin embargo, no es lo mismo escribir un programa en lenguaje de máquina que en un lenguaje de alto nivel, donde el lenguaje mismo es independiente de la máquina. En la actualidad hay tantos lenguajes de programación como formas de afrontar un problema.

Debido a la gran importancia que tienen los lenguajes de programación surge la necesidad de estudiarlos y analizarlos a profundidad para buscar nuevas alternativas para atacar los problemas de implementación y diseño que se presentan en el desarrollo de software, así como el estudiar técnicas que nos permitan saber o demostrar que un programa realiza la tarea que se desea. Es decir, queremos estar seguros que un programa resuelve el problema para el que fue hecho.

2.2. Su evolución

El avance de la computación en nuestros días se da a pasos agigantados, tanto software como hardware avanzan rápidamente dando como resultado nuevas formas de atacar los problemas para ganar velocidad, almacenamiento o facilidad de uso en los programas o en las máquinas mismas.

El avance en el software es muy importante, nos permite atacar nuevos problemas, resolver los ya existentes o hacer las soluciones más eficientes.

Se podría pensar en un principio que el estado de los lenguajes de programación en la actualidad es estable, sin embargo, no es raro encontrar nuevos paradigmas de programación, nuevos enfoques para atacar los problemas, y todo el tiempo surgen nuevas metodologías de programación y con esto nuevos lenguajes.

Para estudiar un lenguaje debemos poner atención en su sintaxis, que los programas que se escriban pertenezcan al lenguaje, es decir, estén bien formados. También debemos estudiar su semántica, es decir, el significado de los programas escritos en el lenguaje.

A continuación estudiaremos cómo es que se debe entender lo que un programa quiere decir para que la máquina lo traduzca a las acciones deseadas. De esta forma, podemos llegar a demostrar propiedades sobre el comportamiento de un programa para un lenguaje dado. Sería interesante tener una forma de demostración que nos permita abarcar no sólo un lenguaje de programación sino un conjunto de éstos. Veremos más adelante que mediante la idea propuesta en el presente trabajo, es posible construir un sistema de demostración formal más general. Pasemos ahora a estudiar cómo podemos entender el comportamiento de un programa.

2.3. La semántica de los lenguajes

La semántica de los lenguajes la podemos entender como el proceso que nos permite conocer el significado de los programas, es decir, interpretar los símbolos que pertenecen al lenguaje y realizar las acciones

correspondientes. De forma análoga a como hacemos con el lenguaje natural, en la semántica de los lenguajes de programación entendemos la semántica como la forma en que traducimos los elementos del lenguaje en objetos que tienen sentido y significado para nosotros.

Para que quede más claro el concepto podemos emplear el siguiente ejemplo:

El cielo es azul

Sabemos que las palabras que forman la oración pertenecen al lenguaje español, es decir, son palabras bien formadas sintácticamente. Sin embargo podemos entender perfectamente qué es lo que estamos diciendo gracias a la semántica del lenguaje, sabemos lo que representa cada palabra, sabemos a qué objetos nos estamos refiriendo. Es decir, el conjunto de palabras tiene sentido.

Algo similar pasa con los lenguajes de programación, con la ventaja de que, a diferencia de las lenguas naturales donde encontramos muchas ambigüedades, en los lenguajes de programación tenemos un esquema más rígido. Sin embargo, la idea es similar, una vez que tenemos un programa sintácticamente válido debemos darnos cuenta si lo que tenemos es algo que tenga sentido, es decir, que tenga una interpretación válida.

Con un ejemplo de un programa muy sencillo podemos terminar de aclarar esta idea:

```
x = (1+4);
```

Podemos decir que este sencillo programa está bien construido sintácticamente pues pertenece a un lenguaje (C por ejemplo), pero de acuerdo a la semántica podemos saber que se trata de una asignación, que debemos hacer una suma de dos enteros y el resultado será guardado en la variable `x`. El símbolo '+' toma el significado de la operación aritmética que corresponde a la suma, los símbolos numéricos 1 y 4 son interpretados como las dos cantidades que representan en la aritmética estándar.

Ahora estudiemos más a detalle estos conceptos.

2.3.1. Importancia de la semántica formal

Los primeros lenguajes de programación (ALGOL 60 por ejemplo) usaban lenguaje natural e informal para describir su semántica. Debido a esto tenían muchos errores y ambigüedades en su definición y las incompatibilidades entre las implementaciones se hicieron inevitables. Es decir, se podían encontrar formas diferentes de interpretar el mismo programa. Fue entonces cuando quedó claro que las especificaciones formales eran importantes. Para tratar de dar especificaciones semánticas se han abordado varias formas de hacerlo.

2.3.2. La semántica operacional

La semántica operacional describe el comportamiento de un lenguaje definiendo inductivamente las relaciones de transición para expresar la evaluación y la ejecución del lenguaje.

Por ejemplo, supongamos en un lenguaje dado consideramos:

- $n, m \in N$
- $X, Y \in Loc$
- $a \in ExpA$
- $b \in ExpB$
- $c \in Com$

donde Loc es el conjunto de localidades de memoria, $ExpA$ es el conjunto de las expresiones aritméticas, $ExpB$ es el conjunto de las expresiones booleanas y Com es el conjunto de comandos del lenguaje.

Estos conjuntos los podemos describir mediante reglas de formación:

Para $ExpA$:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

Para $ExpB$:

$$b ::= \textbf{cierto} \mid \textbf{falso} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

Para *Com*:

$$c ::= \textbf{skip} \mid X := a \mid c_0; c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } c$$

De esta manera tenemos una definición inductiva del lenguaje. A continuación podemos decir cómo se comportan los programas escritos en este lenguaje cuando son ejecutados.

Podemos pensar de forma intuitiva en cómo se evalúan los conjuntos antes mencionados utilizando de alguna forma los estados, o valores de las variables que el programa utiliza. Una expresión aritmética se evalúa en un entero, una expresión booleana se evalúa en un valor de verdad.

Los valores resultantes pueden afectar la ejecución de los comandos, los cuales son el medio para modificar los estados por los que se va pasando al ejecutarse el programa.

El conjunto de estados Σ son funciones $\sigma : Loc \to \mathbb{N}$ de las localidades a los naturales (en el caso particular del lenguaje que estamos manejando). Podemos ahora considerar la evaluación de una localidad en un estado σ dado. Supongamos que a en el estado σ tiene un valor de n, esto es:

$$\langle a, \sigma \rangle \to n$$

Al par $\langle a, \sigma \rangle$ lo llamaremos configuración.

Ahora podemos seguir con la especificación de las relaciones de evaluación con reglas basadas en la sintaxis:

Para evaluar números:

$$\langle n, \sigma \rangle \to n$$

Al evaluar el número n obtenemos n mismo.

Para evaluar localidades:

$$\langle X, \sigma \rangle \to \sigma(X)$$

Al evaluar una localidad obtenemos el valor de su contenido en ese estado.

Para evaluar las sumas:

$$\frac{\langle a_0\rangle \to n_0 \quad \langle a_1,\sigma\rangle \to n_1}{\langle a_0 + a_1,\sigma\rangle \to n} \text{ donde } n \text{ es la suma de } n_0 \text{ y } n_1$$

Para evaluar los productos:

$$\frac{\langle a_0\rangle \to n_0 \quad \langle a_1,\sigma\rangle \to n_1}{\langle a_0 \times a_1,\sigma\rangle \to n} \text{ donde } n \text{ es el producto de } n_0 \text{ y } n_1$$

Estas reglas expresan por ejemplo, que al evaluar la suma de dos expresiones se obtiene el número que corresponde a la suma de los números correspondientes a la evaluación de los sumandos.

De esta forma, cuando se quiere hacer una evaluación de una expresión aritmética a en algún estado σ lo que se hace es buscar una derivación en la cual la parte izquierda de la conclusión case con $\langle a,\sigma\rangle$. A la estructura que vamos construyendo para llegar a tener la evaluación de una expresión la llamamos árbol de derivación.

Aunque esto no pasa para expresiones aritméticas, en general, más de una regla tiene una parte de la izquierda que casa con las configuraciones dadas. Para garantizar el encontrar un árbol de derivación con la conclusión que case, cuando existe una, todas las reglas que casen deben ser consideradas, para ver si pueden ser las conclusiones de las derivaciones. Todas las posibles derivaciones con la conclusión deben ser construidas en paralelo, es decir, se construyen varios árboles de derivación.

De esta forma obtenemos un algoritmo para la evaluación de expresiones aritméticas basado en la búsqueda de un árbol de derivación. Como puede ser implementado casi directamente a partir de las reglas la especificación del significado, es decir, de *la semántica* de las expresiones aritméticas de forma operacional, se dice que estas reglas dan una *semántica operacional* de tales expresiones.

La relación de evaluación determina una relación natural de equivalencia sobre las expresiones, definida como:

$$a_0 \sim a_1 \; sii \; (\forall n \in N \forall \sigma \in \Sigma.\langle a_0,\sigma\rangle \to n \iff \langle a_1,\sigma\rangle \to n),$$

es decir, dos expresiones aritméticas son equivalentes si y solo si al evaluarse producen el mismo valor.

La evaluación de expresiones booleanas

Ahora veamos cómo se evalúan las expresiones booleanas a los valores de verdad con las siguientes reglas:

$$\langle \textbf{cierto}, \sigma \rangle \rightarrow \textbf{cierto}$$

Análogamente para el caso de **falso**.

$$\frac{\langle a_0, \sigma \rangle \rightarrow n \quad \langle a_1, \sigma \rangle \rightarrow m}{\langle a_0 = a_1, \sigma \rangle \rightarrow \textbf{cierto}} \text{ si } n \text{ y } m \text{ son iguales.}$$

Análogamente para el caso de **falso**, en el caso en que no sean iguales obtenemos **falso**.

$$\frac{\langle a_0, \sigma \rangle \rightarrow n \quad \langle a_1, \sigma \rangle \rightarrow m}{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \textbf{cierto}} \text{ si } n \text{ es menor o igual que } m.$$

Análogamente para el otro caso en el que tendremos **falso**.

$$\frac{\langle b, \sigma \rangle \rightarrow \textbf{cierto}}{\langle \neg b, \sigma \rangle \rightarrow \textbf{falso}}$$

Análogamente cuando la evaluación de b es **falso**, tenemos **cierto**.

Ahora vemos el comportamiento para los operadores lógicos:

$$\frac{\langle b_0, \sigma \rangle \rightarrow t_0 \langle b_1, \sigma \rangle \rightarrow t_1}{\langle b_0 \wedge b_1, \sigma \rangle \rightarrow t}$$

donde t es **cierto** si $t_0 \equiv$ **cierto** y $t_1 \equiv$ **cierto** y $t \equiv$ **falso** en otros casos.

Análogamente para $\vee$, si alguno de los dos es **cierto** el resultado es **cierto** y **falso** si los dos lo son.

De esta forma, las reglas nos dicen cómo reducir las expresiones booleanas a un valor de verdad.

Otra vez hay una relación natural de equivalencia en las expresiones booleanas: dos expresiones booleanas son equivalentes sii se evalúan en el mismo valor de verdad en todos los estados:

$$b_0 \sim b_1 \text{ sii } \forall t \forall \sigma \in \Sigma. \langle b_0, \sigma \rangle \rightarrow t \Longleftrightarrow \langle b_1, \sigma \rangle \rightarrow t$$

La ejecución de comandos

Las expresiones producen valores al evaluarse en un estado en particular. Los programas, y por tanto, los comandos, se ejecutan para cambiar el estado. Un par $\langle c, \sigma \rangle$ representa una configuración (o comando), que indica que se ejecuta el comando c sobre el estado σ. Esto es, al ejecutarse c se llevan a cabo posibles cambios en las variables, modificando así el estado σ y teniendo como resultado un nuevo estado σ'. Definimos entonces la relación

$$\langle c, \sigma \rangle \to \sigma'$$

Ahora veamos la notación que usaremos para definir las reglas que describen la ejecución de comandos:

$$\sigma[m/X](Y) = \begin{cases} m & \text{si } Y = X, \\ \sigma(Y) & \text{si } Y \neq X \end{cases}$$

Reglas para los comandos:
Comandos atómicos

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

Secuencial

$$\frac{\langle c_0, \sigma \rangle \to \sigma'' \quad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$$

Condicionales

$$\frac{\langle b, \sigma \rangle \to \mathbf{cierto} \quad \langle c_0, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{falso} \quad \langle c_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, \sigma \rangle \to \sigma'}$$

Ciclos

$$\frac{\langle b, \sigma \rangle \to \mathbf{falso}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{cierto} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \to \sigma'}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma'}$$

De nuevo hay una relación natural de equivalencia sobre comandos:

$$c_0 \sim c_1 \text{ sii } \forall \sigma, \sigma' \in \Sigma. \langle c_0, \sigma\rangle \to \sigma' \Longleftrightarrow \langle c_1, \sigma\rangle \to \sigma'$$

2.4. La semántica denotacional

En la semántica operacional podemos observar que hay ciertas arbitrariedades en las reglas, por ejemplo el tamaño de los pasos de las transiciones de la derivación. Además en la descripción del comportamiento se mezcla la sintaxis, lo que hace difícil comparar dos programas escritos en diferentes lenguajes. También notemos que el estilo de la semántica es bastante cercano a una implementación del lenguaje, por lo que la descripción puede ser dada a un intérprete para conocer la equivalencia entre expresiones aritméticas, booleanas o comandos. Por ejemplo (como veíamos anteriormente):

$$c_0 \sim c_1 \text{ sii } (\forall \sigma, \sigma'. \langle c_0, \sigma\rangle \to \sigma' \Longleftrightarrow \langle c_1, \sigma\rangle \to \sigma').$$

Sin embargo hay otra forma más directa de capturar la semántica del lenguaje si solo estamos interesados en los comandos equivalentes. Notemos que $c_0 \sim c_1$ sii:

$$\{(\sigma, \sigma') | \langle c_0, \sigma\rangle \to \sigma'\} = \{(\sigma, \sigma') | \langle c_1, \sigma\rangle \to \sigma'\}$$

En otras palabras, dos comandos son equivalentes si y solo si ambos determinan la misma función parcial sobre los estados. Esto sugiere la posibilidad de definir el significado o la semántica del lenguaje en una forma más abstracta en la cual tomamos la denotación de un comando haciéndola una función parcial sobre los estados. Esta nueva forma de describir la semántica es lo que conocemos como *semántica denotacional.*

Una expresión aritmética $a \in ExpA$ denotará una función

$$\mathcal{A}[\![a]\!] : \Sigma \to \mathbb{N}$$

Una expresión booleana $b \in ExpB$ denotará una función

$$\mathcal{B}[\![b]\!] : \Sigma \to T$$

del conjunto de estados al conjunto de valores de verdad (es decir: $T =$ {cierto, falso}).

Un comando c denotará una función parcial $\mathcal{C}[\![c]\!] : \Sigma \to \Sigma$.

Notemos que $\mathcal{A}$ es una función de las expresiones aritméticas del tipo $ExpA \to (\Sigma \to N)$, y en lo primero que pensamos cuando vemos una expresión aritmética es en evaluarla. Los corchetes puestos a una expresión quieren decir "no se evalúe" ya que es un objeto sintáctico. Entonces ponemos los corchetes alrededor de una expresión como argumento de una función semántica para mostrar que el argumento es un objeto sintáctico.

Ahora veamos las denotaciones de las expresiones que tenemos:

Para $ExpA$:

$\mathcal{A}[\![n]\!] = \{(\sigma, n) | \sigma \in \Sigma\}$

$\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X)) | \sigma \in \Sigma\}$

$\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) | (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \ \& \ (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$

Análogamente para la resta y la multiplicación.

Nótese que los signos "+", "-", "×" del lado izquierdo de la definición representan símbolos sintácticos mientras que los signos de la derecha representan las operaciones sobre los números, por ejemplo para cualquier estado σ,

$$\mathcal{A}[\![3 \times 5]\!]\sigma = \mathcal{A}[\![3]\!]\sigma \times \mathcal{A}[\![5]\!]\sigma = 3 \times 5 = 15$$

Para $ExpB$:

$$\begin{aligned}
\mathcal{B}[\![\textbf{cierto}]\!] &= \{(\sigma, \textbf{cierto}) | \sigma \in \Sigma\} \\
\mathcal{B}[\![\textbf{falso}]\!] &= \{(\sigma, \textbf{falso}) | \sigma \in \Sigma\} \\
\mathcal{B}[\![a_0 = a_1]\!] &= \{(\sigma, \textbf{cierto}) | \sigma \in \Sigma \ \&\ \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \\
&\quad \{(\sigma, \textbf{falso}) | \sigma \in \Sigma \ \&\ \mathcal{A}[\![a_0]\!] \neq \mathcal{A}[\![a_1]\!]\sigma\} \\
\mathcal{B}[\![a_0 \leq a_1]\!] &= \{(\sigma, \textbf{cierto}) | \sigma \in \Sigma \ \&\ \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\} \cup \\
&\quad \{(\sigma, \textbf{falso}) | \sigma \in \Sigma \ \&\ \mathcal{A}[\![a_0]\!] \not\leq \mathcal{A}[\![a_1]\!]\sigma\} \\
\mathcal{B}[\![\neg b]\!] &= \{(\sigma, \neg t) | \sigma \in \Sigma \ \&\ (\sigma, t) \in \mathcal{B}[\![b]\!]\} \\
\mathcal{B}[\![b_0 \wedge b_1]\!] &= \{(\sigma, t_0 \wedge t_1) | \sigma \in \Sigma \ \&\ (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \ \&\ (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\} \\
&\quad \text{Análogamente para la disyunción de expresiones} \\
&\quad \text{booleanas.}
\end{aligned}$$

Para comandos:

La definición de $\mathcal{C}[\![c]\!]$ para comandos es más complicada. Primero daremos denotaciones como relaciones entre estados; y después seguiremos la inducción estructural para mostrar que son en realidad funciones parciales.

Es bastante claro que:

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!] &= \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![X := a]\!] &= \{(\sigma, \sigma[m/X]) \mid \sigma \in \Sigma \ \&\ m = \mathcal{A}[\![a]\!]\sigma\} \\
\mathcal{C}[\![c_0; c_1]\!] &= \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] &= \{(\sigma, \sigma') | \mathcal{B}[\![b]\!]\sigma = \textbf{cierto} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \cup \\
&\quad \{(\sigma, \sigma') | \mathcal{B}[\![b]\!]\sigma = \textbf{falso} \ \&\ (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}
\end{aligned}$$

Pero encontramos dificultades cuando consideramos la denotación de un ciclo while,

$$w \equiv \textbf{while } b \textbf{ do } c$$

Notemos que la siguiente equivalencia existe:

$$w \sim \textbf{if } b \textbf{ then } c; w \textbf{ else skip}$$

así que la función parcial $C[\![w]\!]$ debería ser igual a la función parcial $C[\![\textbf{if } b \textbf{ then } c \textbf{ else skip}]\!]$. Entonces tenemos

$$\begin{aligned} C[\![w]\!] &= \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{cierto} \ \&(\sigma,\sigma') \in C[\![c;w]\!]\} \cup \\ &\quad \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{falso} \\ &= \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{cierto} \ \&(\sigma,\sigma') \in C[\![w]\!] \circ C[\![c]\!]\} \cup \\ &\quad \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{falso}\} \end{aligned}$$

Escribiendo ϕ en lugar de $C[\![w]\!]$, β en lugar de $\mathcal{B}[\![b]\!]$ y γ en lugar de $C[\![c]\!]$, necesitamos una función parcial ϕ tal que

$$\begin{aligned} \phi &= \{(\sigma,\sigma') \mid \beta(\sigma) = \textbf{cierto} \ \&(\sigma,\sigma') \in \phi \circ \gamma\} \cup \\ &\quad \{(\sigma,\sigma') \mid \beta(\sigma) = \textbf{falso}\} \end{aligned}$$

Pero tenemos ϕ en los dos lados de la ecuación. Necesitamos entonces una forma de solucionar esta ecuación "recursiva". Se define entonces la función Γ donde:

$$\begin{aligned} \Gamma(\phi) &= \{(\sigma,\sigma') \mid \beta(\sigma) = \textbf{cierto} \ \&(\sigma,\sigma') \in \phi \circ \gamma\} \cup \\ &\quad \{(\sigma,\sigma') \mid \beta(\sigma) = \textbf{falso}\} \\ &= \{(\sigma,\sigma') \mid \exists\sigma''.\beta(\sigma) = \ \textbf{cierto} \ \& \ (\sigma,\sigma'') \in \gamma \ \& \ (\sigma'',\sigma') \in \phi\} \cup \\ &\quad \{(\sigma,\sigma') \mid \beta(\sigma) = \textbf{falso}\} \end{aligned}$$

la cual regresa $\Gamma(\phi)$ dada ϕ. Notemos entonces que lo que se busca es un punto fijo ϕ de Γ en el sentido de que

$$\phi = \Gamma(\phi)$$

Podemos considerar a Γ como un operador sobre los conjuntos definidos por las instancias de reglas:[1]

$$\begin{aligned} R &= \{(\{(\sigma'',\sigma')\}/(\sigma,\sigma')) \mid \beta(\sigma) = \ \textbf{cierto} \ \&(\sigma,\sigma'') \in \gamma\} \cup \\ &\quad \{(\emptyset/(\sigma,\sigma')) \mid \beta(\sigma) = \textbf{falso}\} \end{aligned}$$

Sabemos entonces que Γ tiene un punto fijo mínimo definido por fix, de tal forma que:

$$\phi = fix(\Gamma)$$

[1] Revisar [Winskel, 1993] para profundizar sobre este tema.

De esta forma se toma la denotación del programa **while** como este punto fijo mínimo y se puede seguir con la definición de la semántica denotacional de los comandos.

Siguiendo con la definición estructural:

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!] &= \{(\sigma,\sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![X := a]\!] &= \{(\sigma,\sigma[m/X]) \mid \sigma \in \Sigma \ \&\ m = \mathcal{A}[\![a]\!]\sigma\} \\
\mathcal{C}[\![c_0; c_1]\!] &= \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] &= \{(\sigma,\sigma') | \mathcal{B}[\![b]\!]\sigma = \textbf{cierto} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![c_0]\!]\} \\
&\quad \cup \{(\sigma,\sigma') | \mathcal{B}[\![b]\!]\sigma = \textbf{falso} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![c_1]\!]\} \\
\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= fix(\Gamma)
\end{aligned}$$

donde

$$\begin{aligned}
\Gamma(\psi) = \ & \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \textbf{cierto} \ \&\ (\sigma,\sigma') \in \psi \circ \mathcal{C}[\![c]\!]\} \cup \\
& \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \textbf{falso}\}
\end{aligned}$$

De esta forma definimos una denotación de cada comando como una relación entre los estados. Nótese como la definición semántica es composicional en el sentido de que la denotación de un comando es construida a partir de las denotaciones de sus subcomandos inmediatos, debido a la definición estructural.

Cabe mencionar que existe la equivalencia entre las dos semánticas. Consúltese [Winskel, 1993] para más información.

2.5. La semántica axiomática

Ahora vamos a enfocarnos al problema de cómo probar que un programa hace lo que se desea. Lo que queremos tener es un sistema formal para probar propiedades de los programas basado en reglas para cada construcción en el lenguaje que estamos manejando. Estas reglas son las llamadas reglas (o ternas) de Hoare. Originalmente se intentó no solo probar propiedades de los programas sino también dar un método para explicar el significado de las construcciones de programas; el significado de una construcción fue especificado en términos de "axiomas" (o reglas) que dicen cómo probar propiedades de estas construcciones (ciclos,

asignaciones, condicionales, etc). Debido a esto, esta nueva semántica se llama *semántica axiomática.*

Veamos por ejemplo un programa sencillo en el lenguaje que venimos manejando que nos permita calcular la suma de los primeros 20 naturales:

$$S := 0;$$
$$N := 1;$$
$$(\textbf{while } \neg(N = 21) \textbf{ do } S := S + N; N := N + 1)$$

¿Cómo podemos probar que este programa cuando termina deja en S = 20?.

Desde luego, una cosa que se podría hacer es correrlo de acuerdo a la semántica operacional y ver qué obtenemos. Pero supongamos que el programa es alterado un poco, en lugar de "**while** $\neg(N = 21)$ **do** ...", ponemos "**while** $\neg(N = P + 1)$ **do** ..."y asignamos un valor positivo arbitrario a P antes de empezar. De esta forma podría hacerse la suma hasta cualquier número natural. Sin embargo, ya no podemos probar que el programa es correcto simplemente corriéndolo de acuerdo a la semántica operacional para todos los valores posibles de P. Debemos ahora tener una abstracción y usar alguna lógica para razonar sobre el programa.

Quisiéramos tener un sistema formal de demostración para probar propiedades de los programas basado en reglas para cada construcción posible en el lenguaje que estamos manejando.

La base del sistema de prueba es el invariante: una proposición que es cierta antes, durante y después de la ejecución de un comando. Tenemos entonces un sistema de prueba basado en proposiciones de la forma:

$$\{A\}c\{B\}$$

donde A y B son proposiciones lógicas como las que vimos para $ExpB$ y c es un comando.

Esto quiere decir, cualquier terminación de c de un estado que satisface A termina con un estado que satisface B. A es la precondición y B la poscondición.

Introduciendo algo de notación, decimos que un estado σ satisface la proposición A, o A es verdadera (**cierto**) en el estado σ como :

$$\sigma \models A$$

Como un comando c denota una función parcial de los estados iniciales a los estados finales, la corrección parcial indica:

$$\forall \sigma.(\sigma \models A \ \& \ C[\![c]\!]\sigma \text{ está definido }) \Rightarrow C[\![c]\!]\sigma \models B$$

En caso de que $C[\![c]\!]\sigma$ no esté definido tomamos la convención de $\perp$ como el estado indefinido $C[\![c]\!]\sigma = \perp$ y asumimos que $\perp \models A$ para toda proposición A. Para ser consistentes consideramos que $\perp$ no está en el conjunto de estados Σ.

Podemos describir el significado de $\{A\}c\{B\}$ como:

$$\forall \sigma \in \Sigma.\sigma \models A \implies C[\![c]\!]\sigma \models B$$

Las proposiciones que nos interesan acerca de los programas pertenecen al cálculo de predicados, es decir, queremos tener herramientas para inferir y demostrar propiedades de los programas, para esto debemos usar variables enteras que podamos cuantificar. Extendemos pues a las expresiones aritméticas como el conjunto $ExpAv$:

$$a ::= n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

donde

$n \in N$ (los naturales)
$X \in Loc$ (las localidades)
$i \in VarEnt$ (el conjunto de las variables enteras cuantificables)

Y extendemos las expresiones booleanas para incluir estas expresiones aritméticas más generales con cuantificadores e implicación:

$$P ::= \textbf{cierto} \mid \textbf{falso} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid P_0 \wedge P_1 \mid P_0 \vee P_1 \mid \neg P \mid P_0 \Rightarrow P_1 \mid \forall i.P \mid \exists i.P$$

Llamaremos a este conjunto de expresiones booleanas extendidas como *Prop*.

Nosotros tenemos un sentido común para entender las expresiones y las proposiciones. Sin embargo, como queremos hacer razonamientos sobre los sistemas de demostración basados en proposiciones, no solo en ejemplos, debemos ser más formales y dar una teoría del significado de las expresiones y las proposiciones con variables enteras.

Este tema va más allá del alcance de la presente tesis, pero se encuentra en cualquier texto de lógica. Basta con mencionar que el significado de las nuevas expresiones aritméticas que incluyen a las variables enteras depende de la interpretación que se le de a las variables. Es decir, una interpretación es una función $I : VarEnt \to \mathbb{N}$.

Podemos entonces definir una función semántica $\mathcal{P}$ la cual regresa el valor asociado a una expresión aritmética con variables enteras en un estado y una interpretación particulares. El valor de una expresión $a \in Prop$ para una interpretación I y un estado σ se denota como $\mathcal{P}[\![a]\!]I\sigma$. Definimos entonces esta función por inducción estructural como:

$$\begin{aligned}
&\mathcal{P}[\![n]\!]I\sigma = n \\
&\mathcal{P}[\![X]\!]I\sigma = \sigma(X) \\
&\mathcal{P}[\![i]\!]I\sigma = I(i) \\
&\mathcal{P}[\![a_0 + a_1]\!]I\sigma = \mathcal{P}[\![a_0]\!]I\sigma + \mathcal{P}[\![a_1]\!]I\sigma
\end{aligned}$$

análogamente para las demás operaciones aritméticas.

Lo mismo pasa para la noción de satisfacción que veníamos manejando, esta noción, dependerá ahora de la interpretación particular que estemos considerando. Por ejemplo, definimos algunas reglas por inducción estructural sobre las preposiciones, para todo $\sigma \in \Sigma$ y para una interpretación I:

$$\begin{aligned}
&\sigma \models^I \textbf{cierto} \\
&\sigma \models^I (a_0 = a_1) \quad \text{si} \quad \mathcal{P}[\![a_0]\!]I\sigma = \mathcal{P}[\![a_1]\!]I\sigma \\
&\sigma \models^I A \wedge B \quad \text{si} \quad \sigma \models^I A \;\; \& \;\; \sigma \models^I B \\
&\sigma \models^I A \Rightarrow B \quad \text{si} \quad (\not\models \sigma^I A) \;\; o \;\; \sigma \models^I B \\
&\sigma \models^I \forall i.A \quad \text{si} \quad (\models^{I[n/i]} A \text{ para toda } n \in \mathbb{N}
\end{aligned}$$

análogamente para las demás expresiones.

Es importante decir que esta idea se emplea más adelante en el capítulo donde se revisa una aplicación concreta y se demuestra la corrección de un programa. Sin embargo, para entender la clase de lenguajes sobre la que se trabajará, es necesario revisar otra forma de cómputo distinta a la secuencial.

2.6. El caso paralelo

Sabemos que el tamaño y complejidad de las aplicaciones actuales muchas veces sobrepasan las capacidades en cuanto a tiempo de respuesta de las máquinas tradicionales y a veces es mucho más rápido o eficiente realizar tareas en paralelo, es decir, repartir el trabajo en tareas que se pueden llevar a cabo al mismo tiempo y así realizar el trabajo en menos tiempo. Esto mismo se puede realizar mediante hardware y software, empleando varios procesadores o lenguajes paralelos.

La concurrencia en un lenguaje de programación y el paralelismo en el hardware son conceptos independientes. Las operaciones en hardware ocurren en paralelo si se traslapan en tiempo, esto es, si se ejecutan al mismo tiempo de forma independiente pero controlada. Las operaciones en el código fuente son concurrentes si pueden, pero no necesariamente, ser ejecutadas en paralelo. Podemos tener concurrencia en el lenguaje sin un hardware paralelo, y también tener ejecución paralela sin concurrencia en el lenguaje.

A continuación hablaremos más acerca de los lenguajes paralelos y la concurrencia.

2.7. Introducción al paralelismo y no determinismo

La computación paralela se basa en la subdivisión de una tarea computacional en diversas subtareas que se realizan simultáneamente. Entre las motivaciones centrales de la computación paralela están:

- el incremento de la rapidez de cómputo (es decir, dos computadoras pueden –en principio– hacer en la mitad de tiempo la misma tarea

que una computadora sola);

- mayor eficiencia en el manejo de los recursos computacionales, pues en muchas ocasiones el que una computadora se ocupe de una sola tarea a la vez implica la subutilización de recursos (dispositivos periféricos, memoria, etc.) que podrían emplearse de manera simultánea;
- responder a una serie de problemas de la realidad en los que existe la interacción simultánea de distintos procesos.

La subdivisión de tareas se realiza por medio de una combinación de *hardware* y *software* específicos. Sin embargo, en la mayor parte de los casos se cuenta con un lenguaje de programación en el que se especifica la división de tareas.

Una manera sencilla de abordar el tema del paralelismo es extender nuestro lenguaje del principio con una operación de composición paralela. Para los comandos c_0, c_1, su composición paralela $c_0 \parallel c_1$ se ejecuta como c_0 y c_1 juntos, sin ninguna preferencia.

Podemos explicar la ejecución de la composición paralela de dos comandos por las reglas:

$$\frac{\langle c_0, \sigma\rangle \to_1 \sigma'}{\langle c_0 \parallel c_1, \sigma\rangle \to_1 \langle c_1, \sigma'\rangle} \qquad \frac{\langle c_0, \sigma\rangle \to_1 \langle c_0', \sigma'\rangle}{\langle c_0 \parallel c_1, \sigma\rangle \to_1 \langle c_0' \parallel c_1, \sigma'\rangle}$$

$$\frac{\langle c_1, \sigma\rangle \to_1 \sigma'}{\langle c_1 \parallel c_1, \sigma\rangle \to_1 \langle c_0, \sigma'\rangle} \qquad \frac{\langle c_1, \sigma\rangle \to_1 \langle c_1', \sigma'\rangle}{\langle c_0 \parallel c_0, \sigma\rangle \to_1 \langle c_0 \parallel c_1', \sigma'\rangle}$$

La simetría en las reglas introduce una incertidumbre sobre el comportamiento de los comandos. Consideremos por ejemplo la ejecución del programa $(X := 0 \parallel X := 1)$. Sin importar el estado inicial, no sabemos con qué valor terminará X. Esta incertidumbre sobre el comportamiento es llamada *no determinismo*.

Paradójicamente, un uso disciplinado del no determinismo nos puede llevar a una presentación más sencilla de los algoritmos. Esto es porque en algunos casos el objetivo de un programa podría no depender de cuál tarea se debe realizar primero.

Para lograr esto debemos extender nuestro lenguaje con dos nuevos conjuntos de comandos y de comandos con guardias. La sintaxis de los nuevos comandos está dada por las reglas:

$$c ::= \textbf{skip} \mid \textbf{abort} \mid X := a \mid c_0; c_1 \mid \textbf{if } cg \textbf{ fi} \mid \textbf{do } cg \textbf{ od}$$

$$cg := b \rightarrow c \mid cg_0 [\!] cg_1$$

El constructor usado para formar comandos con guardia es llamado alternativa. El comando con guardia típico tiene la forma

$$(b_1 \rightarrow c_1) [\!] \ldots [\!] (b_n \rightarrow c_n)$$

En este contexto las expresiones booleanas b son llamados los guardias, la ejecución del comando c_i depende de que el guardia correspondiente b_i se evalúe a **cierto**. Si ningún guardia resulta verdadero en una estado el comando con guardia falla, en este caso el comando no termina en un estado final. De otra forma, el comando ejecuta de manera no determinista los comandos c_i cuyo guardia b_i sea verdadero. El nuevo comando **abort** no termina en un estado final desde cualquier estado inicial. El comando **if** cg **fi** se ejecuta como cg si no falla, y de otra forma, se comporta como **abort**. El comando **do** cg **od** ejecuta repetidamente el comando cg hasta que éste falle y actúa como **skip** si cg falla inicialmente.

Capturamos esta explicación informal mediante las siguientes reglas operacionales:

$$\langle \mathbf{skip}, \sigma \rangle \to \sigma$$

$$\frac{\langle a, \sigma \rangle \to n}{\langle X := a, \sigma \rangle \to \sigma[n/X]}$$

$$\frac{\langle c_0, \sigma \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \langle c_1, \sigma' \rangle} \qquad \frac{\langle c_0, \sigma \rangle \to \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \to \langle c_0'; c_1, \sigma' \rangle}$$

$$\frac{\langle cg, \sigma \rangle \to \langle c, \sigma' \rangle}{\langle \mathbf{if}\ cg\ \mathbf{fi}, \sigma \rangle \to \langle c, \sigma' \rangle}$$

$$\frac{\langle cg, \sigma \rangle \to \mathbf{fail}}{\langle \mathbf{do}\ cg\ \mathbf{od}, \sigma \rangle \to \sigma} \qquad \frac{\langle cg, \sigma \rangle \to \langle c, \sigma' \rangle}{\langle \mathbf{do}\ cg\ \mathbf{od}, \sigma \rangle \to \langle c; \mathbf{do}\ cg\ \mathbf{od}, \sigma' \rangle}$$

Reglas para los comandos con guardias:

$$\frac{\langle b, \sigma \rangle \to \mathbf{cierto}}{\langle b \to c, \sigma \rangle \to \langle c, \sigma \rangle}$$

$$\frac{\langle cg_0, \sigma \rangle \to \langle c, \sigma' \rangle}{\langle cg_0 [\!] cg_1, \sigma \rangle \to \langle c, \sigma' \rangle} \qquad \frac{\langle cg_1, \sigma \rangle \to \langle c, \sigma' \rangle}{\langle cg_0 [\!] cg_1, \sigma \rangle \to \langle c, \sigma' \rangle}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{falso}}{\langle b \to c, \sigma \rangle \to \mathbf{fail}} \qquad \frac{\langle cg_0, \sigma \rangle \to \mathbf{fail} \quad \langle cg_1, \sigma \rangle \to \mathbf{fail}}{\langle cg_0 [\!] cg_1, \sigma \rangle \to \mathbf{fail}}$$

Un ejemplo sería un comando que asigne el máximo valor de dos localidades X y Y a la localidad MAX:

$$\begin{array}{l} \mathbf{if} \\ X \geq Y \to MAX := X \\ [\!] \\ Y \geq X \to MAX := Y \\ \mathbf{fi} \end{array}$$

La simetría entre X y Y se perdería en el lenguaje tradicional que veníamos manejando.

Con los programas concurrentes parece natural permitir decisiones arbitrarias en el cómputo, siempre y cuando esto no afecte el resultado, dejando de lado el determinismo.

Ahora pongamos atención en un formalismo que nos permita modelar el comportamiento descrito anteriormente en otros términos. Empezaremos por entender los sistemas de reducción.

Capítulo 3

Sistemas de Reducción Compuestos

Para entender los Sistemas de reducción compuestos (SRC), estudiamos primero los sistemas de reducción para luego abordar el tema de la composición de estos sistemas.

3.1. Importancia de los SRC

Los SRC pueden entenderse como sistemas de reescritura de términos, es decir, se tienen reglas reescritura que nos permiten transformar los elementos del conjunto que estemos tratando. Un sistema de reducción es simplemente una forma más abstracta de presentar a los sistemas de reescritura de términos. En el caso de los sistemas de reducción compuestos se tienen operadores de composición sobre las reglas de reescritura, el operador de composición paralela y el de composición secuencial.

El concepto de un sistema de reescritura de términos (SRT) es paradigmático para el estudio de los sistemas de cómputo. Ya hace medio siglo, el cálculo lambda, probablemente el mejor conocido de los SRT, tuvo un papel importante en la lógica matemática con respecto a formalizar la noción de computabilidad; mucho después, el mismo SRT figuró en el trabajo fundamental de Scott, Plotkin y otros, llevándolos a un nuevo enfoque en la semántica denotacional de lenguajes de programación. Recientemente, este sistema de lógica combinatoria, mostró ser una fructífera herramienta para la implementación de lenguajes funcionales. Más recientemente otra familia de SRT realizó importantes

conexiones entre los conceptos de teoría de categorías y los pasos elementales en los cálculos computacionales.

Los SRT son atractivos por su sintaxis y semántica sencillas —al menos los SRT que no involucran variables acotadas como el cálculo lambda, pero involucran la reescritura de términos de un lenguaje de primer orden. Este aspecto facilita un análisis matemático satisfactorio. Por otro lado, proporcionan un medio natural para implementar el cómputo, y en principio aun para el cómputo paralelo. Esta característica hace interesantes a los SRT para el diseño de máquinas de reducción paralela.

Otro campo donde los SRT tienen un papel fundamental es en análisis e implementación de especificaciones de tipos de datos abstractos (propiedades de consistencia, teoría de computabilidad, decibilidad de problemas, demostración de teoremas).

3.2. Los sistemas de reducción

Los sistemas de reducción son simplemente conjuntos con una relación binaria (o colección de relaciones) de "re-escritura" como mencionábamos anteriormente. Un sistema de reducción puede pensarse como una forma abstracta de cómputo que abarca los conceptos fundamentales de la computación: *iteración, terminación y no-terminación.*

El cómputo puede verse como el proceso de re-escritura que hemos mencionado de forma iterativa, empezando con algún objeto del conjunto, y la terminación corresponde a obtener un objeto que no puede ser reescrito más. Por otro lado, la no-terminación podemos verla como la capacidad de reescribir indefinidamente.

Ahora veamos algunas definiciones[1]. de las cuales hablaremos más cuando veamos el caso concreto de Gamma con SRC.

Sea $\mathscr{S} = \langle M, \rightarrow \rangle$ un sistema de reducción, donde M es el conjunto de elementos y $\rightarrow$ es la relación de reescritura.

- La cerradura transitiva y reflexiva de $\rightarrow$ se denota $\rightarrow^*$. Entonces $a \rightarrow^* b$ si hay una sucesión finita posiblemente vacía de "pasos

[1]Puede revisarse [Abrahamsky and Gabbay, 1990] para profundizar en el tema

de reducción" $a \equiv a_0 \to a_1 \to \cdots \to a_n \equiv b$. La relación de equivalencia generada por $\to$ es $=$, también llamada relación de conversión.

- Decimos que $a \in M$ es una forma normal si para ninguna $b \in M$ se tiene que $a \to b$. Es decir, si a ya no se puede reescribir más.
- $\mathscr{S}$(o $\to$, denotando al sistema simplemente por su relación) tiene una forma normal única si $\forall a, b \in M (a = b \wedge a, b$ son formas normales $\Rightarrow a \equiv b)$.
- $\mathscr{S}$(o $\to$) tiene la propiedad de forma normal si $\forall a, b \in M$ (a es forma normal y $a = b \Rightarrow b \to^* a$).

3.2.1. Algunos ejemplos de Sistemas de Reducción

Podemos ver algunos ejemplos de sistemas de reducción:
El cálculo lambda es claramente un sistema de reducción, por ejemplo considerando la siguiente expresión:

$$\lambda func.\lambda arg.(func\ arg)$$

que corresponde a la aplicación de funciones, veamos cómo se realiza la reducción. La relación de reescritura corresponde a la reducción β por ejemplo:

$$\lambda func.\lambda arg.(func\ arg)\ sen\ \pi \to_\beta \lambda arg.(sen\ arg)\ \pi \to_\beta sen\ \pi$$

También podemos ver a una gramática que define un lenguaje formal como un sistema de reducción donde la relación de reescritura es simplemente la aplicación de las producciones:

$$\begin{array}{lcl} A & := & N + N \mid N - N \mid N * N \mid N/N \\ N & := & 0 \mid suc(N) \end{array}$$

Se puede pensar entonces en tener composiciones de estos sistemas con el fin de tener construcciones más complejas.

3.3. Los Sistemas de Reducción Compuestos (o programas)

Consideraremos los sistemas (o "programas") cuyos componentes básicos son las relaciones de reducción de algunos sistemas de reducción. Estos sistemas, llamados sistemas de reducción compuestos, se construyen componiendo relaciones de reducción con dos operadores naturales de composición: la composición *paralela* y *la secuencial*.[2].

Veamos formalmente la idea anterior mediante la semántica para los SRC.

3.3.1. Sintaxis y semántica operacional de los SRC

Un sistema de reducción es una pareja $\langle \mathbf{T}, \{\rightarrow_r\}_{r\in R}\rangle$ donde $\mathbf{T}$ es un conjunto con elementos $M, N, M_1 \ldots$ (también llamados términos o estados). Las relaciones de reducción $\{\rightarrow_r\}_{r\in R}$ son relaciones binarias sobre estados. Estas relaciones binarias sobre los estados serán las unidades básicas de los sistemas de reducción compuestos.

Sea $\langle \mathbf{T}, \{\rightarrow_r\}_{r\in R}\rangle$ un SRC. Para alguna $r \in R$ (alguna relación de un SRC) y $M, N \in \mathbf{T}$ decimos que:

- M se reduce a N si $M \rightarrow_r N$ (i.e. $(M, N) \in \rightarrow_r$)
- M converge inmediatamente, escrito $M \downarrow^r$, si $\neg\exists N \in \mathbf{T}. M \rightarrow_r N$

Los sistemas compuestos de reducción sobre P, Q, P_1, Q_1, etc. están dados por la siguiente gramática:

$$P ::= r \mid P; Q \mid P \parallel Q$$

Un SRC puede estar dado por la relación de reescritura (r), por la composición secuencial de dos relaciones ($P; Q$) o por la composición paralela de dos relaciones ($P \parallel Q$).

También podemos nombrar a los SRC como programas para fines prácticos como se ha hecho entender con la definición anterior de la gramática.

[2]Tomado del artículo de Sands, "Composed Reduction Systems" [Sands, 1996]

Los SRC o sistemas de reescritura se pueden entender mediante una función $RT : SRC \times T \rightarrow \mathbb{P}(T)$ definida de las parejas de SRC y términos de los SRC (reglas de reescritura y los conjuntos que maneja el SRC) a los términos de los SRC como sigue:

$$RT(R,t) = \{s \mid t \rightarrow_R s\}$$

Esta función RT describe todos los términos que se pueden alcanzar a partir de un término aplicando las reglas de reescritura.

Hay algunas cosas que se deben entender primero para familiarizarse con los nuevos conceptos y entender el comportamiento de los SRC.

- La composición paralela permite la combinación arbitraria de pasos de reducción. En el caso más simple, la composición paralela de dos relaciones de reducción es la unión de las relaciones. La composición paralela termina cuando, ambos subsistemas terminan simultáneamente.
- La composición secuencial nos lleva fuera del campo de los sistemas reducción (sobre el conjunto dado). La composición secuencial de dos relaciones de reducción es el sistema que se comporta como la primer relación, hasta la terminación del primer sistema, después se comporta como el segundo sistema. Se dice que termina cuando el segundo subsistema ha terminado.

Aclarando las ideas anteriores: los sistemas reducción compuestos no son necesariamente sistemas de reducción, pero tienen una noción de "paso de reducción", y una noción correspondiente a terminación. Vamos a aclarar mejor este punto porque servirá para entender el por qué la composición secuencial no corresponde a una composición relacional de las relaciones de reducción.

Los SRC no son necesariamente sistemas de reducción. Para que esto fuera cierto, el resultado de la composición (secuencial y paralela) de relaciones sería una relación.

La composición paralela puede ser modelada por una relación, en particular, la relación es la unión de las relaciones que son compuestas de forma paralela. Sin embargo, esto no pasa cuando para la composición

secuencial de dos relaciones (la composición secuencial de dos relaciones no corresponde a la composición relacional de éstas).

Para la composición secuencial de dos relaciones R_1 y R_2, las parejas que aparecen en R_1 dependen del estado inicial, es decir, dependen de los pasos anteriores en el sentido de que se debe conocer en qué momento ha terminado la aplicación de R_1 para poder aplicar R_2. Esto no puede ser modelado mediante una relación pues sólo podemos distinguir que una pareja pertenece a una relación o a otra pero no cuándo ha concluido una para poder aplicar la otra.

Supongamos que tenemos la relación que es la composición relacional (o también llamado el producto relativo)[3] de las relaciones que se quieren componer secuencialmente (denotaremos la composición relacional como $R_1 \bullet R_2$). Sabemos que la composición secuencial se comporta como R_1 hasta que no se puede aplicar mas, es decir, a partir de un elemento obtenemos otro mediante R_1 y con el elemento obtenido hacemos lo mismo y así sucesivamente hasta terminar (en caso de que sea posible) cuando R_1 no se puede aplicar más, obtenemos otro elemento (a partir del último obtenido con R_1) aplicando R_2 hasta que no se puede aplicar más.

La composición relacional es:

$$\forall x \forall y((x,y) \in R_1; R_2 \leftrightarrow \exists z((x,z) \in R_1 \wedge (z,y) \in R_2))$$

donde R_1, R_2 son las relaciones en cuestión y x, y, z son elementos del conjunto.

Ahora veamos un ejemplo para darnos cuenta que la composición relacional y secuencial pueden no ser iguales:

Sean $T = \{a,b,c,d,e,f,s\}$, $R_1 = \{(a,b),(b,c),(c,d)\}$, $R_2 = \{(d,e),(e,f),(b,s)\}$.

Notemos que para la composición secuencial tenemos que:

$$\text{por un lado} \qquad a \xrightarrow[R_1]{*} d$$

$$\text{por otro lado} \qquad d \xrightarrow[R_2]{*} f$$

[3]Véase [Tarski, 1941] Tarski, 1941 donde se trata del cálculo de relaciones.

entonces tenemos con la composición secuencial que

$$a \xrightarrow[R_1;R_2]{*} f$$

Ahora veamos qué pasa con la composición relacional. Como en la composición relacional podemos aplicar en cualquier momento la relación R_2, podemos tener:

$$a \xrightarrow[R_1 \bullet R_2]{*} s$$

Por lo tanto sabemos que un SRC no siempre es un sistema de reducción. Notemos entonces que es necesario tener alguna forma de describir el comportamiento que se observa en la composición secuencial. Pasamos entonces a una descripción más formal del comportamiento de los SRC.

3.3.2. Semántica Operacional Estructural

Debido a la presencia de la composición secuencial, los programas no pueden ser vistos como sistemas de reducción sobre los términos, ya que el programa no es una entidad estática (como ya se explicó anteriormente). Para definir la semántica para estos programas definimos una relación transición de un sólo paso entre configuraciones. Las configuraciones son pares programa-estado, escritos $\langle P, M\rangle$. El resultado final de un cálculo está dado por un predicado de convergencia inmediata, $\downarrow$, sobre configuraciones.

Las reducciones de un solo paso y la convergencia inmediata están dadas por la siguientes reglas:

$$\frac{M \to_{\mathrm{r}} N}{\langle \mathrm{r}, M\rangle \to \langle \mathrm{r}, N\rangle} \qquad \frac{M\downarrow^{\mathrm{r}}}{\langle \mathrm{r}, M\rangle \downarrow} \qquad \frac{\langle P, M\rangle \downarrow \quad \langle Q, M\rangle \downarrow}{\langle P \parallel Q, M\rangle \downarrow}$$

$$\frac{\langle P, M\rangle \to \langle P', M'\rangle}{\langle P; Q, M\rangle \to \langle P'; Q, M'\rangle} \qquad \frac{\langle P, M\rangle \downarrow}{\langle P; Q, M\rangle \to \langle Q, M\rangle}$$

$$\frac{\langle P, M\rangle \to \langle P', M'\rangle}{\langle P \parallel Q, M\rangle \to \langle P' \parallel Q, M'\rangle \quad \langle Q \parallel P, M\rangle \to \langle Q' \parallel P, M'\rangle}$$

Trivialmente podemos verificar que la convergencia inmediata de una configuración corresponde a la ausencia de alguna transición para esa configuración. Es decir, $\langle P, M\rangle \rightarrow \langle Q, N\rangle$ para algún $\langle Q, N\rangle$ si y solo si $\neg(\langle P, M\rangle \downarrow)$.

3.3.3. Semántica denotacional para los SRC

Ahora veamos una forma de describir el comportamiento de un SRC no como una serie de pasos (semántica operacional) sino mediante "objetos" matemáticos que podemos manejar.

Sea D un conjunto arbitrario. Consideremos ahora el siguiente conjunto:

$$\mathbb{T}(D) \;=\; \{(M_1, M_2)(M_3, M_4)\ldots | M_i \in D\}$$

cuyos elementos llamaremos trazas. $\mathbb{T}_{fin}(D)$ denotará el conjunto de trazas finitas, y ϵ denotará la traza vacía.

Ahora veamos algunas operaciones y relaciones sobre trazas que serán útiles más adelante.

Concatenación: $\mathbb{T}_{fin}(D) \times \mathbb{T}(D) \rightarrow \mathbb{T}(D)$. Si $\alpha \in \mathbb{T}_{fin}(D), \beta \in \mathbb{T}(D)$ su concatenación se denota por $\alpha\beta$.

Liga. $\odot : \mathbb{T}_{fin}(D) \times \mathbb{T}(D) \rightarrow \mathbb{T}(D)$. Ligar dos trazas difiere de la concatenación como muestra la siguiente definición:

$$(M_1, M_2)\ldots(M_{n-1}, M_n) \odot (N_1, N_2)\ldots = \begin{cases} (M_1.\, M_2)\ldots(M_{n-1}, M_n)(N_1, N_2)\ldots & \text{si } M_n = N_1 \\ indefinido & \text{de otra forma} \end{cases}$$

Absorción. $A \subset \mathbb{T}(D) \times \mathbb{T}(D)$. Sea $\alpha \in \mathbb{T}_{fin}(D)$ y $\beta \in \mathbb{T}(D)$. Entonces se cumple:

$$(\alpha(M, N)(N, P)\beta, \alpha(M, P)\beta) \in A$$

Si $T \subset \mathbb{T}(D)$ entonces $\bar{A}(T)$ será su cerradura bajo absorción.

Absorción total. $\_ : \mathbb{T}(D) \rightarrow \mathbb{T}(D)$. El resultado de la absorción total es el último elemento en la cadena $t_1, \ldots, t_n, \ldots t_\alpha$ tal que $A(t_i, t_{i+1})$ y $t_i \neq t_{i+1}$. Nótese que la cadena puede ser infinita con un elemento que sirva de cota. La absorción total $(\underline{\ })$ es calculada de forma recursiva de acuerdo a la siguiente regla:

$$\begin{aligned} \underline{\epsilon} &= \epsilon \\ \underline{(M_1, M_2)(M_3, M_4)\alpha} &= \begin{cases} \underline{(M_1, M_4)\alpha} & \text{si } M_2 = M_3 \\ (M_1, M_2)\underline{(M_3, M_4)\alpha} & \text{de otra forma} \end{cases} \end{aligned}$$

Los conjuntos de trazas pueden entenderse como denotaciones de programas en lenguajes paralelos con estados. Cada par (M_i, M_{i+1}) en una traza denota una *transición* del estado M_i al estado M_{i+1} generado por cierto programa.

En el caso de los SRC, cada par (M_i, M_{i+1}) en una traza denota una *transición* del término M_i al término M_{i+1} de acuerdo a la regla de reescritura del SRC en cuestión.

En otras palabras, la denotación de un SRC la entendemos como las posibles trazas producidas a partir de la aplicación de las reglas de reescritura del SRC.

3.4. Lógica para los SRC

Como se menciona en capítulos anteriores, se busca una forma de expresar propiedades acerca de los SRC, así como demostrar dichas propiedades. Una vez que se tenga una manera formal de hacer esto, se buscará transformar los programas de Gamma en SRC para tener forma de probar corrección sobre dichos sistemas.

A continuación se presentan algunas herramientas lógicas sobre los SRC. Estas herramientas servirán para probar propiedades de los SRC. (Esto se aplicará en el capítulo §6 a un caso real, cuando se tenga la transformación antes mencionada de los programas de Gamma a SRC).

3.4.1. El lenguaje lógico para los SRC: $L(SRC)$.

La forma en que se pueden expresar propiedades acerca de los SRC, así como demostrarlas formalmente, es mediante una lógica sobre dichos sistemas.

Sea $S = \langle T, R\rangle$ un SRC. Definimos un lenguaje lógico $L(T)$ sobre los términos T de S, el cual nos permita la construcción de proposiciones sobre los términos del SRC. Este lenguaje será empleado para expresar las condiciones necesarias para la aplicación de las reglas de reescritura de un SRC.

Suponemos entonces la existencia de una relación de satisfacción $\models_T \subseteq T \times L(T)$ para términos T de un SRC:

$$t \models_T \psi \quad \text{sii} \quad \begin{array}{l} t \text{ cumple con la propiedad } \psi, \text{ es decir, } t \text{ se puede} \\ \text{transformar aplicando alguna de las reglas de } R \end{array}$$

donde $t \in T$ y $\psi \in L(T)$ es alguna condición necesaria para aplicar alguna de las reglas de reescritura R del SRC.

Veamos un ejemplo para que esto sea más claro:

Sea S un sistema de reescritura dado por una gramática definida por las siguientes producciones:

$$aAb \rightarrow aaab$$
$$aBb \rightarrow abbb$$

donde el conjunto de términos para este sistema son las cadenas α sobre el alfabeto Σ donde $\Sigma = \{a, b\} \cup \{A, B\}$.

Ahora definimos un lenguaje $L(T) = \{\psi_1, \psi_2\}$ donde

$$\begin{aligned} \psi_1 &= \text{`}\alpha \text{ tiene la forma } aAb\text{'} \\ \psi_2 &= \text{`}\alpha \text{ tiene la forma } aBb\text{'} \end{aligned}$$

Ahora tomemos algunos términos del sistema S y veamos qué pasa con la relación de satisfacción que definimos. Sea $\alpha_1, \alpha_2 \in \Sigma^*$ donde $\alpha_1 = aaA$ y $\alpha_2 = aAb$, ahora consideremos la relación de satisfacción:

$$\alpha_1 \not\models \psi_1$$
$$\alpha_2 \models \psi_1$$

de esta forma sabemos que α_2 puede transformarse de acuerdo a las reglas de reescritura del sistema S pues cumple con la propiedad ψ_1 que expresa la condición para que pueda aplicarse la primera producción de la gramática que habíamos definido.

De acuerdo a la semántica para los SRC dada anteriormente, la denotación de un SRC la podemos entender por medio de trazas sobre los términos del SRC (como habíamos mencionado anteriormente). Entonces un SRC o programa se denota por la transformación de un término en otro a través de las reglas de reescritura, y de esta forma se producen las trazas que describen la aplicación de estas reglas. Veamos ahora la manera en que formalizamos esta idea:

Sea $S = \langle T, R\rangle$ un SRC y sea $\mathcal{T}(T)$ el conjunto de las trazas sobre T. Definimos ahora un lenguaje lógico $L(SRC)$ que nos permita expresar propiedades sobre los SRC. La sintaxis de $L(SRC)$ se define por la siguiente regla:

$$\frac{\bar{\alpha} \in \mathcal{T}(T)}{\diamond\bar{\alpha} \in L(SRC)}$$

La forma en que se expresan las propiedades sobre los SRC usando este lenguaje es siguiendo la idea intuitiva de la denotación de un SRC por medio de trazas.

Definimos una relación de satisfacción $\models_{SRC} \subseteq SRC \times L(SRC)$ de manera inductiva usando las siguientes reglas:[4]

Sea $S = \langle T, R\rangle$ un SRC.

$$\frac{t \models_T \psi_S, \quad t' \in RT(R,t), \quad S \models_{SRC} \diamond\bar{\alpha}}{S \models_{SRC} \diamond(t,t') \odot \bar{\alpha}}\ Mediador \qquad \frac{t \not\models \psi_S}{S \models_{SRC} \diamond(t,t)}\ Terminal$$

donde $t, t' \in T, \psi \in L(T)$ y $\bar{\alpha} \in \mathcal{T}(T)$. En la regla *Mediador*, $t' \in RT(R,t)$ se puede entender como $t \underset{R}{\rightarrow} t'$, es decir, t' se puede obtener a partir de t aplicando alguna regla del SRC.

[4]Las reglas que aquí se proponen fueron presentadas en [Hernández, 1999] para multiconjuntos.

Ahora veamos qué pasa para los operadores:

Composición secuencial

$$\frac{P \models_{SRC} \diamond\bar{\alpha}, \quad Q \models_{SRC} \diamond\bar{\beta}}{P; Q \models_{SRC} \diamond\bar{\alpha} \odot \bar{\beta}}$$

Composición paralela I

$$\frac{P \models_{SRC} \diamond\bar{\alpha}\bar{\beta}, \quad P' \models_{\mathcal{T}} \diamond\bar{\beta}, \quad P' \parallel Q \models_{SRC} \diamond\bar{\gamma}}{P \parallel Q \models_{SRC} \diamond\bar{\alpha} \odot \bar{\gamma}}$$

Composición paralela II

$$\frac{Q \models_{SRC} \diamond\bar{\alpha}\bar{\beta}, \quad Q' \models_{SRC} \diamond\bar{\beta}, \quad P \parallel Q' \models_{SRC} \diamond\bar{\gamma}}{P \parallel Q \models_{SRC} \diamond\bar{\alpha} \odot \bar{\gamma}}$$

donde $P, P', Q \in SRC$ y $\bar{\alpha}, \bar{\beta}, \bar{\gamma} \in \mathcal{T}(T)$.

Ahora el significado intuitivo de la relación de satisfacción para los SRC es claro y usando estas reglas se puede probar formalmente propiedades sobre los SRC como veremos más adelante.

Estas reglas fueron propuestas originalmente para una lógica sobre multiconjuntos, sin embargo es aplicable también para SRC por la transformación propuesta en esta tesis, de tal forma que las reglas de inferencia son correctas y condicionalmente completas. Las demostraciones que aparecen en [Hernández, 1999] siguen siendo válidas para este trabajo.

Ahora hablemos del caso que nos importa, un lenguaje de naturaleza paralela, Gamma.

Capítulo 4

Gamma

Gamma es un lenguaje de programación que está basando en la metáfora de una reacción química. La idea es tener una solución química que contiene moléculas que reaccionan mientras se cumplen ciertas condiciones, dando paso a nuevos componentes en la solución. Las reacciones dejan de ocurrir cuando ya ninguna molécula puede reaccionar en las condiciones dadas.

4.1. Acerca de Gamma

El formalismo de Gamma fue propuesto hace más de diez años e intenta modelar una solución química de moléculas donde éstas interactúan libremente. Fue introducido entonces el multiconjunto para simular la solución química y un programa para simular una reacción, formada a su vez por una dupla *(condición, acción)*. La ejecución de un programa significa reemplazar las moléculas que están reaccionando en la solución por el resultado de dicha reacción. Esto es, los elementos del multiconjunto que cumplen la condición de reacción son sustituidos por el producto de la acción. El programa termina cuando se alcanza un estado estable, es decir, cuando ninguna reacción puede llevarse a cabo bajo la condición de reacción dada.

Gamma posee un alto nivel de abstracción, debido a las pocas restricciones impuestas por su único tipo de datos. Debido a esto, Gamma puede ser usado como lenguaje intermedio en el paso de derivar un programa a partir de su especificación.

Debido a la naturaleza de Gamma, se presta mucho para programar aplicaciones en paralelo donde se deba tener un lenguaje de coordinación para describir las interacciones entre las entidades de la aplicación.

Sin embargo, al estudiar más a fondo a Gamma podemos encontrar que presenta algunas debilidades. Por ejemplo, la definición original de Gamma no permite combinar programas de forma clara (véase [De La Cruz Martínez, 2002] donde se trata el tema más a fondo mediante el estudio de Gamma de orden superior). Además, por la explosión combinatoria que surge de la semántica (cuando definimos las composiciones paralelas y en la verificación de las condiciones de reacción) es difícil pensar en una implementación sencilla y eficiente del lenguaje. También hay que tomar en cuenta que el lenguaje no hace fácil para el programador el estructurar datos, o especificar estrategias de control. Este último punto será tratado en mayor detalle más adelante cuando pongamos atención en Gamma estructurado.

4.2. Conceptos básicos

Aunque decíamos que las bases para comprender y estudiar Gamma son muy sencillas, hay algunas entidades que debemos conocer.

4.2.1. Multiconjuntos

Los multiconjuntos pueden ser pensados como una colección de elementos donde éstos pueden repetirse. Formalmente un multiconjunto es una función $M : G \to \mathbb{N}$. Es decir, a cada elemento de G se le asocia el número de apariciones en el multiconjunto.

En el presente trabajo se estudian los multiconjuntos finitos. Se trata con multiconjuntos finitos por ser una estructura con las mínimas restricciones, es decir, buscamos manejar un dato estructurado muy general y nos atenemos al hecho de que la memoria de la máquina es finita.

También se usa la siguiente notación para multiconjuntos:

$$\{\!|x_1, \ldots, x_n|\!\}$$

donde los x_i no son necesariamente distintos.

Ahora daremos una noción más detallada de los multiconjuntos y las operaciones con las que vamos a tratar. El conjunto de multiconjuntos finitos de elementos en G se denota como $\mathbb{M}(G)$. Si M y $N \in \mathbb{M}(G)$, definimos la unión, diferencia e intersección mediante las siguientes funciones:

$$(M \uplus N)(x) = M(x) + N(x)$$

$$(M - N)(x) = \begin{cases} M(x) - N(x) & \text{si M(x)} \geq \text{N(x)} \\ 0 & \text{de otra forma} \end{cases}$$

$$(M \cap N)(x) = min\{M(x), N(x)\}$$

Análogamente si $M(x) \geq N(x) \quad \forall x \in G$, podemos decir que $N(x) \subseteq M(x)$.

4.2.2. Sintaxis y semántica

Intuitivamente, los programas de Gamma son hechos de reacciones atómicas (que como veíamos para los SRC, se pueden considerar reglas de reescritura), las cuales toman un multiconjunto, verifican que ciertas condiciones para la reacción se cumplan y transforman el multiconjunto de acuerdo a las reglas correspondientes a las acciones.

Las reacciones atómicas se pueden componer en forma secuencial o paralela.

Ahora veamos cómo se define y comporta Gamma de manera más formal:

$$P ::= (x_1, \ldots, x_n) \rightarrow A^n(x_1, \ldots, x_n) \Leftarrow R^n(x_1, \ldots, x_n) \mid P \circ P \mid P \parallel P,$$

donde R^n es un predicado (condición) y A^n es una acción. $P \circ P$ es la composición secuencial de programas de Gamma y $P \parallel P$ es la composición paralela de programas de Gamma.

El efecto de una reacción atómica en un multiconjunto M es tomar una tupla que satisfaga R^n y reemplazarla con el resultado de aplicar A^n a la misma tupla. Si no existe tal tupla, entonces el multiconjunto M queda sin cambios y la reacción atómica termina.

Para que el comando $P_2 \circ P_1$ termine, primero se ejecuta P_1 hasta que termine y luego debe terminar P_2. Para que $P_1 \parallel P_2$ termine los dos deben terminar al mismo tiempo.

Una regla de reescritura $A \Leftarrow R$ puede ser considerada una función de los multiconjuntos a las conjuntos de multiconjuntos transformados. Para entender este punto de vista, vamos a introducir otra función $S : \mathbb{A}^n \times R^n \times \mathbb{M}(T) \to \mathscr{P}_{fin}(\mathbb{M}(T))$ definida como sigue:

$$S(A^n, R^n, M) \;=\; \{N \mid N = (M - \{\!|x_1, \ldots, x_n|\!\}) \uplus A^n(x_1, \ldots, x_n)\}$$

donde $\{\!|x_1, \ldots, x_n|\!\}$ corresponde a la tupla que satisface la condición expresada en R^n.

Esta nueva función la podemos entender como una analogía con la función RT que tenemos para la transformación de términos de acuerdo con la regla de reescritura (definida en §3.3.1).

Esto será formalizado a continuación mediante una semántica ope-

racional.[1]

$$\frac{\{\!|a_1,\ldots,a_n|\!\} \subseteq M, \quad R(a_1,\ldots,a_n)}{\langle (A \Leftarrow R), M\rangle \rightarrow \langle (A \Leftarrow R), (M - \{\!|a_1,\ldots,a_n|\!\}) \uplus A(a_1,\ldots,a_n)\rangle}$$

$$\frac{\neg\exists \{\!|a_1,\ldots,a_n|\!\} \subseteq M.R(a_1,\ldots,a_n)}{\langle (A \Leftarrow R), M\rangle \rightarrow M}$$

$$\frac{\langle Q, M\rangle \rightarrow M}{\langle P \circ Q, M\rangle \rightarrow \langle P, M\rangle} \qquad \frac{\langle Q, M\rangle \rightarrow \langle Q', M'\rangle}{\langle P \circ Q, M\rangle \rightarrow \langle P \circ Q', M'\rangle}$$

$$\frac{\langle P, M\rangle \rightarrow \langle P', M'\rangle}{\langle P \parallel Q, M\rangle \rightarrow \langle P' \parallel Q, M'\rangle} \qquad \frac{\langle Q, M\rangle \rightarrow \langle Q', M'\rangle}{\langle P \parallel Q, M\rangle \rightarrow \langle P \parallel Q', M'\rangle}$$

$$\frac{\langle P, M\rangle \rightarrow M \quad \langle Q, M\rangle \rightarrow M}{\langle P \parallel Q, M\rangle \rightarrow M}$$

Notemos que esta definición de la semántica operacional induce una "semántica de un solo paso", es decir, los efectos de las diferentes reacciones compuestas de forma paralela ocurren solo una al mismo tiempo. En principio este tipo de semántica no modela el paralelismo de los programas de Gamma, sin embargo esta semántica es equivalente a la "semántica de pasos múltiples"[2] en la cual las diferentes reacciones pueden actuar al mismo tiempo sobre subconjuntos independientes de un multiconjunto dado. Esto es claramente la forma de simular el paralelismo de Gamma. Sin embargo es empleada la semántica operacional de un solo paso porque es más sencilla y fácil de tratar.

[1] Presentada también en Hankin et al. (1993)

[2] Esto es explicado a detalle y demostrado en [Chaudron, 1998]

4.2.3. Algunos ejemplos

Un programa para calcular el producto de los dos números más grandes en un multiconjunto, es el siguiente:

$$Max = (x, y, z) \rightarrow \{\!|x, y|\!\} \Leftarrow x \geq z \wedge y \geq z$$
$$Prod = (x, y) \rightarrow \{\!|x \cdot y|\!\} \Leftarrow cierto$$
$$P = Prod \circ Max$$

Un programa que calcula el n-ésimo número de Fibonacci aplicado al multiconjunto $\{\!|n|\!\}$, Propuesto originalmente por [Hankin C., 1998].

$$Pred = x \rightarrow \{\!|x-1, x-2|\!\} \Leftarrow (x > 1)$$
$$Uno = x \rightarrow \{\!|1|\!\} \Leftarrow (x = 0)$$
$$Sum = (x, y) \rightarrow \{\!|x+y|\!\} \Leftarrow cierto$$
$$Fib = Sum \circ (Pred \parallel Uno)$$

Estos ejemplos fueron tomados de la tesis [Hernández, 1999], ahí se demuestra la corrección de estos programas.

4.3. Gamma estructurado

Es claro que en Gamma es difícil hablar de cosas como datos estructurados o tipos debido a que en realidad sólo se tienen tipos básicos y multiconjuntos de éstos. Debido a esto, no es muy sencillo intentar una estrategia de control de tipos ya que la motivación original del lenguaje fue precisamente el poder describir programas con las mínimas restricciones. Una consecuencia desfavorable es que en estos casos el programador tiene que utilizar "trucos" para expresar algún algoritmo.

La dificultad de estructurar en Gamma se presenta entonces en los programas y en la implementación de éstos.

Debido a esto, surgió Gamma estructurado[3]. La idea básica es la noción de multiconjunto estructurado, el cual es un conjunto de direc-

[3]Presentada en el artículo "Structured Gamma" de Fradet y D. Le Métayer [P. Fradet, 1998]

ciones asociadas con valores. Estas direcciones deben cumplir algunas relaciones específicas.

Como un ejemplo, la lista [1;2;3] puede ser representada por un multiconjunto estructurado cuyo conjunto de direcciones es $\{a_1, a_2, a_3\}$ y sus valores asociados (denotados como $\overline{a_i}$) son: $\overline{a_1} = 1, \overline{a_2} = 2, \overline{a_3} = 3$. Ahora, para tener el orden en la lista tomamos una relación **next**. Sea **next** una relación binaria y **end** una relación unaria; las direcciones satisfacen

$$\textbf{next}\ a_1\ a_2, \textbf{next}\ a_2\ a_3, \textbf{end}\ a_3$$

Las relaciones expresan la forma en que se comporta una vecindad de moléculas de la solución. Un tipo es definido en términos de reglas de reescritura sobre las relaciones del multiconjunto. Más adelante aclararemos estas nociones.

4.3.1. Sintaxis de Gamma estructurado

La sintaxis de Gamma estructurado es descrita por la siguiente gramática:

$$\begin{aligned}
\langle Programa \rangle &::= NomPrograma = [\langle Reaccion \rangle]^* \\
\langle Reaccion \rangle &::= \langle Accion \rangle \Longleftarrow\!\!| \langle Condicion \rangle \\
\langle Condicion \rangle &::= \mathbf{r}\ x_1 \cdots x_n \mid f^{Bool}(\bar{x}_1, \ldots, \bar{x}_n) \mid \\
&\quad \langle Condicion \rangle, \langle Condicion \rangle \\
\langle Accion \rangle &::= \mathbf{r}\ x_1 \cdots x_n \mid x := f^{v}(\bar{x}_1, \ldots, \bar{x}_n) \mid \\
&\quad \langle Accion \rangle, \langle Accion \rangle
\end{aligned}$$

donde $NomPrograma$ es simplemente una etiqueta correspondiente al nombre del programa, $\mathbf{r}$ ($\in \mathbb{R}$) denota una relación binaria, x_i es una variable de dirección, $\bar{x}_i$ es el valor de la dirección x_i y f^X es una función de $\mathbb{V}$ a X.

Además, para que este diseño funcione, un programa de Gamma estructurado debe satisfacer las siguientes condiciones sintácticas:

- Si $\bar{x}$ ocurre en la reacción entonces x ocurre en la condición
- Una acción no puede tener dos asignaciones a la misma variable

4.3.2. La semántica de gamma estructurado

Denotamos con A(M) el conjunto de direcciones que ocurren en el multiconjunto M y con $\uplus$ la unión de multiconjuntos. Un multiconjunto estructurado M puede ser visto como $M = Rel \uplus Val$ donde

- *Rel* es el multiconjunto de relaciones representadas como tuplas $(\mathrm{r}, a_1, \ldots, a_n)$ (con $\mathrm{r} \in R$ y $a_i \in \mathrm{A}$).
- *Val* es el conjunto de valores representado por las tripletas de la forma (val, a, v) (con $a \in \mathbf{A}$ y $v \in V$).

Un multiconjunto estructurado válido es tal que una dirección x no tiene más de un valor (es decir, x ocurre a lo más una vez en *Val*). Por otro lado, puede haber varias ocurrencias de la misma tupla en *Rel*. Nótese que no es forzoso que:

$$\mathrm{A}(Rel) \subseteq \mathrm{A}(Val)$$
$$\mathrm{A}(Val) \subseteq \mathrm{A}(Rel)$$

Para definir la semántica de los programas, asociamos tres funciones con cada reacción $(A \Longleftarrow C)$:[4].

$$\begin{aligned}
\mathscr{T}(C)(a_1 \ldots, a_i, b_1, \ldots, b_j) &= (\mathbf{val}, a_1, \bar{a}_1) \in Val \wedge \cdots \wedge \\
&\quad (\mathbf{val}, a_i, \bar{a}_i) \in Val \wedge (\mathbf{val}, b_1, \bar{b}_1) \in Val \\
&\quad \wedge \cdots \wedge (\mathbf{val}, b_j, \bar{b}_j) \in Val \wedge \lfloor C \rfloor \\
\mathscr{C}(C)(a_1 \ldots, a_i, b_1, \ldots, b_j) &= \{(\mathbf{val}, a_1, \bar{a}_1) \cdots (\mathbf{val}, a_i, \bar{a}_i), (\mathbf{val}, b_1, \bar{b}_1), \\
&\quad \cdots, (\mathbf{val}, b_j, \bar{b}_j)\} + \lceil C \rceil \\
\mathscr{A}(A)(a_1, \ldots, a_i, b_1 \ldots, b_j, c_1, \ldots, c_k) &= \{(\mathbf{var}, a_1, \bar{a}_1), \cdots, (\mathbf{var}, a_i, \bar{a}_i)\} \uplus \lceil A \rceil
\end{aligned}$$

[4]la semántica también es presentada en el artículo "Structured Gamma" de Fradet y D. Le Métayer [P. Fradet, 1998]

donde $\lfloor\ \ \rfloor$ está definido por:

$$\begin{aligned} \lfloor X_1, X_2 \rfloor &= \lfloor X_1 \rfloor \wedge \lfloor X_2 \rfloor \\ \lfloor \mathbf{r} x_1 \cdots x_n \rfloor &= (\mathbf{r}, x_1, \ldots x_n) \in \mathit{Rel} \\ \lfloor f(\bar{x}_1, \ldots, \bar{x}_n) \rfloor &= f(\bar{x}_1, \ldots, \bar{x}_n) \end{aligned}$$

y $\lceil\ \ \rceil$ está definido por:

$$\begin{aligned} \lceil X_1, X_2 \rceil &= \lceil X_1 \rceil \uplus \lceil X_2 \rceil \\ \lceil \mathbf{r} x_1 \cdots x_n \rceil &= \{(\mathbf{r}, x_1, \ldots x_n)\} \in \mathit{Rel} \\ \lceil f(\bar{x}_1, \ldots, \bar{x}_n) \rceil &= \emptyset \\ \lceil x := f(\bar{x}_1, \ldots, \bar{x}_n) \rceil &= \{(\mathbf{val}, x, f(\bar{x}_1, \ldots, \bar{x}_n))\} \end{aligned}$$

y

- $\{a_1, \ldots, a_i\}$ denota el conjunto de variables no asignadas cuyos valores ocurren en la relación,
- $\{b_1, \ldots, b_j\}$ denota el conjunto de variables asignadas que ocurren en la condición C,
- $\{c_1, \ldots, c_k\}$ denota el conjunto de variables que ocurren solamente en la acción A.

La función booleana $\mathscr{T}(C)$ representa la condición de aplicación de la reacción. La función $\mathscr{C}(C)$ representa las tuplas seleccionadas por la condición (es decir, las relaciones y valores que aparecen en C). La función $\mathscr{A}(A)$ representa las tuplas agregadas por la acción, esto es: las relaciones que ocurren en A, los valores seleccionados pero sin cambio por la reacción y los valores asignados.

Entonces la semántica de un programa en Gamma estructurado

$$P = [(A_1 \Longleftarrow C_1), \ldots, (A_m \Longleftarrow C_m)]$$

aplicado a un multiconjunto M está definida como el conjunto de formas normales del siguiente sistema de reescritura.

$$\begin{aligned} M \quad &\longrightarrow_P \quad \mathscr{G}\mathscr{C}(M) \\ &\qquad \text{si } \forall \{x_1, \ldots, x_n\} \subseteq \mathrm{A}(M) \forall i \in [1, \ldots, m]. \neg \mathscr{T}(C_i)(x_1, \ldots, x_n). \\ M \quad &\longrightarrow_P \quad M - \mathscr{C}(C)(x_1, \ldots, x_n) \uplus \mathscr{A}(A)(x_1, \ldots, x_n, y_1, \ldots, y_k) \\ &\qquad \text{con } y_1, \ldots, y_k \notin \mathrm{A}(M) \\ &\qquad \text{y } \{x_1, \ldots, x_n\} \subseteq \mathrm{A}(M), i \in [1, \ldots, m] \text{ y } \mathscr{T}(C_i)(x_1, \ldots, x_n) \end{aligned}$$

Entonces podemos describir el multiconjunto del ejemplo de la sección §4.3 en nuestro primer ejemplo de multiconjuntos estructurados, el multiconjunto estructurado puede ser escrito como

$$\{(\mathbf{next}, a_1, a_2), (\mathbf{next}, a_2, a_3), (\mathbf{end}, a_3), (\mathbf{val}, a_1, 1), (\mathbf{val}, a_2, 2), (\mathbf{val}, a_3, 3)\}$$

Si ninguna n-ada de direcciones satisface alguna condición entonces se llega a una forma normal. El resultado es la estructura accesible descrita por las relaciones, es decir, corresponde a llevar una reacción: se toman los elementos en el multiconjunto que cumplen la condición y son reemplazados por los nuevos elementos que son resultado de llevar a cabo la acción correspondiente.

La función $\mathscr{GC}$ quita de Val las direcciones que no ocurren en Rel. Formalmente

$$\mathscr{GC}(Rel \uplus Val) \quad = \quad Rel \uplus \{(\mathbf{val}, a, v) | (\mathbf{val}, a, v) \in Val \wedge a \in \mathrm{A}(Rel)\}$$

Por otro lado, una tupla de direcciones $(x_1, \ldots, x_n)$ y un par (C_i, A_i) tales que $\mathscr{T}(C_i)(x_1 \ldots, x_n)$ son elegidos de forma no determinista. El multiconjunto es transformado quitando $\mathscr{C}(C_i)(x_1, \ldots, x_n)$, asignando nuevas variables $y_1, \ldots, y_k$ y agregando $\mathscr{A}(A_i)(x_1, \ldots, x_n, y_1, \ldots, y_k)$.

4.3.3. Ejemplo

Ahora veamos un ejemplo de un programa en Gamma y su equivalente en Gamma estructurado.

Un programa en Gamma estructurado está definido en términos de pares formados por una condición y una acción tales que

- verifican / modifican las relaciones sobre las direcciones,
- verifican / modifican los valores asociados con las direcciones

Usando las relaciones que definimos en el primer ejemplo de la sección §4.3 para multiconjuntos estructurados podemos programar un algoritmo de ordenamiento.

El algoritmo escrito en Gamma para ordenar un multiconjunto es el siguiente

$$\begin{aligned} Sort \;=\; & [((\mathbf{val}, x, \bar{x}), (\mathbf{val}, y, \bar{y}), (\mathbf{next}, x, y)) \rightarrow \\ & (\mathbf{next}, x, y), (\mathbf{val}, x, \bar{y}), (\mathbf{val}, y, \bar{x}) \Longleftarrow\!| (\bar{x} > \bar{y})] \end{aligned}$$

y en Gamma estructurado tenemos este programa

$$Sort = [\mathbf{next}\ x\ y, \bar{x} > \bar{y} \Longmapsto \mathbf{next}\ x\ y,\ x := \bar{y},\ y := \bar{x}]$$

las dos direcciones seleccionadas x y y deben satisfacer la relación **next** $x\ y$ y sus valores $\bar{x}$ y $\bar{y}$ son tales que $\bar{x} > \bar{y}$. La acción de intercambia sus valores y deja la relación como estaba. Pero para que tenga sentido el programa, el multiconjunto reescrito por *Sort* debe ser del tipo *List* y la relación preserve el tipo del multiconjunto. Estos puntos los veremos a continuación.

4.3.4. Los tipos estructurados

Los tipos estructurados pueden ser vistos como una técnica sintáctica que le sirve al programador para hacer explícita la organización de los datos. Ahora introducimos una nueva notación para caracterizar la estructura de un multiconjunto. Definimos un tipo en términos de reglas de reescritura sobre las relaciones del multiconjunto. Un multiconjunto estructurado se dice que pertenece a un tipo si su conjunto de direcciones puede producirse por el sistema de reescritura que define el tipo.

La sintaxis de los tipos está definida por la siguiente gramática:

$$\begin{aligned} \langle DeclTipo \rangle &::= NomTipo = \langle Prod \rangle, [\langle NoTerm \rangle = \langle Prod \rangle]^* \\ \langle NoTerm \rangle &::= NoTermNom\ x_1, \ldots, x_n \\ \langle Prod \rangle &::= \mathbf{r}\ x_1, \ldots, x_n \mid \langle NoTerm \rangle \mid \langle Prod \rangle, \langle Prod \rangle \end{aligned}$$

donde *NomTipo* y *NoTermNom* simplemente son etiquetas correspondientes al nombre del tipo y nombre de los símbolos no terminales

respectivamente, r ($\in R$) es una relación n-aria (n > 0), y x_i es una variable de direcciones.

Ejemplo:

$$List = L\ x$$
$$L\ x = \textbf{next}\ x\ y, L\ y$$
$$L\ x = \textbf{end}\ x$$

Es una definición para las listas.

En el artículo "Structured Gamma" de Fradet y Le Métayer [P. Fradet, 1998] se presenta un algoritmo para realizar la verificación de tipos y se muestra su corrección. En capítulos posteriores se hablará con mayor detalle de este tema y la forma en que es aplicado en el presente trabajo.

Capítulo 5

La generalización del formalismo Gamma

Nuestra intención es buscar una generalización del modelo de Gamma y Gamma estructurado en un SRC que nos de un modelo más estudiado y más general que permita trabajar no sólo con programas de Gamma o Gamma estructurado para verificar propiedades o hacer demostraciones sobre éstos sino con una subclase de lenguajes paralelos (los que se puedan transformar en SRC).

Mediante la transformación propuesta en este trabajo podemos tener una muy buena idea de cómo se puede realizar esta transformación para otros lenguajes de programación.

5.1. Introducción

La semántica tiene un papel muy importante en la verificación y transformación de programas (como se revisa en §2.3). En términos generales, la verificación de programas tiene como objetivo la demostración de que un programa es correcto por métodos matemáticos precisos. La transformación de programas, por su parte, crea técnicas para desarrollar versiones más eficientes de un programa dado sin alterar la semántica del programa, pues de este modo se asegura que el programa sigue siendo correcto o como proponemos a continuación, la transformación se lleva a cabo como una generalización de los programas de

Gamma en SRC.

Por supuesto, la verificación y transformación de programas son fundamentales en el desarrollo de software confiable, ya sea éste de naturaleza secuencial o paralela.

Debido a que los lenguajes que pertenecen al grupo de los SRC han sido estudiados a profundidad[1] y conocemos su semántica y métodos para demostrar su corrección, queremos llevar los programas de Gamma y Gamma estructurado a SRC para poder probar propiedades empleando reglas sobre la semántica de los SRC como se verá más adelante en el capítulo 6.

De esta forma se busca tener una forma de tener demostraciones más generales sobre propiedades para la clase de lenguajes que se puedan transformar en SRC.

5.2. Transformando Gamma

Para ver un programa de Gamma (o de Gamma estructurado) como un sistema de reducción compuesto simplemente basta con definir el conjunto de estados T ya que el conjunto de relaciones R es el programa mismo.

$$(M, M') \in r \iff M \rightarrow_p M'$$

donde p es el programa de Gamma (o de Gamma estructurado) en cuestión y $R = \{r\}$.

El conjunto de estados se puede ver como el conjunto de multiconjuntos (o multiconjuntos estructurados) ya que la regla de reescritura se puede ver como la aplicación del programa p que transforma un multiconjunto M (o multiconjunto estructurado) en otro multiconjunto M'.

Un ejemplo de esta sencilla transformación para un programa de Gamma estructurado podría ser:

Sea *Sort* el programa estructurado tal que

$$Sort \;=\; [\textbf{next}\;\; x\; y, x := \bar{y},\; y := \bar{x} \Longleftarrow \textbf{next}\;\; x\; y, \bar{x} > \bar{y}]$$

[1] Véase [Sands, 1996]

Y sea M el multiconjunto estructurado que estamos considerando tal que :

$$M = \{(\mathbf{next}, a_1, a_2), (\mathbf{next}, a_2, a_3), (\mathbf{next}, a_3, a_4), \ldots, (\mathbf{next}, a_{n-1}, a_n), (\mathbf{val}, a_1, x_1), (\mathbf{val}, a_2, x_2), \ldots, (\mathbf{val}, a_n, x_n), (\mathbf{end}, a_n)\}$$

Este programa puede verse como un SRC de la siguiente forma:

El conjunto de estados (o términos) T (siguiendo con la definición de un SRC que se dio en §3.3) será el conjunto de multiconjuntos estructurados, donde cada multiconjunto tiene la forma de M, ya que en las reglas de escritura solo modificamos las parejas que pertenecen a la relación **val**. Entonces los multiconjuntos serán muy parecidos a M solo que con algunos de los valores para las direcciones modificados.

Las relaciones de reducción serían la aplicación del programa mismo:

$$(M, M') \in r \iff M \rightarrow_{Sort} M'$$

donde M y M' son multiconjuntos estructurados miembros de T y $R = \{r\}$.

De esta forma tenemos el conjunto de estados como el conjunto de multiconjuntos estructurados y el conjunto de relaciones estará dado por los diferentes programas de Gamma estructurado en cuestión.

Nótese entonces que al transformar programas de Gamma estructurado a SRC (lo mismo se aplica para programas de Gamma a SRC, dando la misma relación de reescritura y el conjunto de multiconjuntos) no cambiamos las propiedades fundamentales del programa (sino simplemente la sintaxis) y las propiedades semánticas se conservan, por lo tanto las propiedades y los algoritmos de verificación de tipos que Fradet y Le Métayer proponen en su artículo "Structured Gamma" [P. Fradet, 1998] siguen siendo válidas.

De esta forma, podemos ver los programas de Gamma y Gamma estructurado como SRC. Esta generalización nos puede ayudar en la verificación de programas usando las reglas que se proponen cuando se estudia la semántica de los SRC (en §3.3.2) y de esta forma tener demostraciones más generales.

Trabajar sobre cosas más generales (como los SRC) nos puede llevar a tener un aparato semántico que nos permita hacer demostraciones

sobre los programas que pertenezcan a la clase de lenguajes que se pueden transformar en SRC como se estudia en el capítulo siguiente.

Capítulo 6

Una aplicación sencilla: "Protocolos de coherencia"

La aplicación que se presenta a continuación está basada en el artículo "Formalization and Verification of Coherence Protocols with the Gamma Framework", de David Mentré, Daniel Le Métayer y Thierry Priol [D. Le Métayer, 1997].

6.1. Motivación y objetivos

Los sistemas distribuidos, son muy usados actualmente. Sin embargo, diseñar e implementar tales sistemas es una difícil tarea que contempla el manejo de paralelismo explícito y duplicación de la información que se maneja. La coherencia es uno de los problemas más difíciles que aparecen con los sistemas distribuidos.

Como en los sistemas distribuidos tenemos la información muy dispersa en todo el sistema, se necesitan protocolos para tener acceso y actualizar la información de forma coherente con restricciones específicas, como tolerancia a fallas, desempeño, tiempo de vida, seguridad, etc.

Los protocolos de coherencia son presentados usando pseudo-código normalmente o lenguaje natural. Esta forma de describir los protocolos dificulta mucho el trabajo de verificación (en caso de que se quisiera probar alguna propiedad o corrección sobre el sistema en cuestión). Sin

embargo, un enfoque más formal nos obligaría a tener la especificación formal y el modelo del protocolo establecidos claramente, esto permite realizar la verificación contra su especificación de manera más sencilla.

Para realizar la descripción del protocolo, se usará un lenguaje de dominio específico. Esta idea está basada en el formalismo Gamma. Además, se propone un algoritmo que está basado en una representación simbólica del sistema, dando como resultado una condición que tiene que ser satisfecha por el estado inicial del sistema y por los estados alcanzados durante su ejecución, es decir, lo que se tiene es un invariante.

6.2. Memorias virtuales compartidas

Las memorias virtuales compartidas (MVC) han sido propuestas por Li y Hudak ([Li and Hudak, 1986])como una técnica para facilitar el diseño de aplicaciones distribuidas. Este concepto ofrece la ilusión de un espacio global de direcciones sobre el espacio de direcciones distribuido. En este sistema, el espacio de direcciones compartido es dividido en unidades elementales llamadas *líneas de caché* (en hardware, memoria distribuida compartida) *o páginas* (en software, MVC). Esta distribución permite el acceso simultáneo a los datos y por tanto, se pueden modificar de manera concurrente varias copias del mismo fragmento lógico de información. Si un nodo del sistema distribuido modifica su copia local de un fragmento de información, las otras copias del mismo fragmento de información deberían ser modificadas. Esto debe hacerse de manera coherente, de tal forma que siempre cada vista del sistema sea coherente (de acuerdo a cierto modelo de memoria, por ejemplo un contrato entre el programador y la MVC). Esta modificación coherente de los datos distribuidos se logra por medio de un protocolo de coherencia de caché.

Para implementar un protocolo, la mayoría del software de MVC usa el mecanismo de paginación de procesadores comunes para detectar el acceso a los datos. En cada nodo, la unidad elemental para un espacio de dirección es una *página* (generalmente de 4 a 8 Kb). Los accesos de lectura y escritura para una página pueden ser puestos de forma

independiente. Si un sistema de MVC debe detectar, por ejemplo, las instrucciones de carga del procesador sobre una página específica, sólo necesita desactivar el acceso de lectura a esta página mediante la lógica de paginación del procesador. Enseguida, el procesador atenderá una instrucción de carga para un dato en esta página, una falla de lectura de página será disparada y atrapada por el sistema de MVC. Cada página es asociada con una página marco que tiene datos en la memoria física.

Mostraremos como ejemplo, el protocolo de coherencia de Li y Hudak de un-escritor/varios-lectores. Este protocolo toma cada página de memoria del espacio de direcciones compartido de manera independiente. Una página es escrita por un solo nodo o leída por uno o más nodos. Cada nodo de lectura tiene una copia local de la página. Cuando una página en estado de lectura es escrita, todas las copias de la página deben ser borradas (mediante la fase de invalidación).

Ahora describimos el formalismo y lo ilustramos con el protocolo antes mencionado.

6.3. Formalización del protocolo

El lenguaje de especificación para el protocolo está basado en el formalismo de Gamma. Se estudia entonces la forma de expresar el protocolo como el sistema de reescritura de multiconjuntos.

6.4. Expresión el protocolo como reescritura de multiconjuntos

El multiconjunto representa una vista global del estado del sistema. En este multiconjunto, las relaciones $\mathbf{R}\ x_1, \ldots, x_n$ describen las entidades del protocolo o los eventos (por ejemplo, para un protocolo de MVC, los derechos de acceso a las páginas, páginas-marco, fallas de escritura o lectura de página, etc.). La lógica del protocolo es descrita por un programa de Gamma hecho de reglas (como $\mathbf{R}_1\ x, \mathbf{R}_2\ y \rightarrow \mathbf{R}_3\ x\ y$). Si el lado izquierdo de la regla se cumple en el multiconjunto, entonces

el lado derecho de la regla ($\mathbf{R}_3 x\ y$) se agrega al multiconjunto. Un paso de reescritura se considera atómico.

Un protocolo *Prt* es descrito por la siguiente gramática:

$$\begin{aligned} Prt &::= Rl \mid Rl, Prt \\ Rl &::= T_1, \quad \neg T_2 \rightarrow T_3 \\ T_i &::= \mathbf{R}\ x_1, \ldots, x_n \mid T_i, T_i \mid \emptyset \end{aligned}$$

Un protocolo está hecho de un conjunto de reglas *Rl*, cada regla tiene tres conjuntos de términos T_1, T_2, T_3. Un conjunto de términos T_i está hecho de cero o más relaciones $\mathbf{R}\ x_1, \ldots, x_n$. En una regla, T_1 es el conjunto que debe estar presente en el multiconjunto para aplicar esta regla. T_2 es el conjunto de relaciones que no deben estar en el multiconjunto para aplicar esta regla. Cuando la regla es aplicada, todas las relaciones de T_1 son eliminadas del multiconjunto y las relaciones de T_3 son agregadas.

A continuación se presenta la especificación completa en Gamma del protocolo:

R_1 : **DetecLec** $p\ \ n_1$, **LMode** $p\ \ n_2$, **Página-Marco** $pf_2\ \ p\ \ n_2$
$\rightarrow$ **LMode** $p\ \ n_1$, **LMode** $p\ \ n_2$, **Página-Marco** $pf_2\ \ p\ \ n_1$
Página-Marco $pf_2\ \ p\ \ n_2$

R_2 : **DetecLec** $p\ \ n_1$, **LEMode** $p\ \ n_2$, **Página-Marco** $pf_2\ \ p\ \ n_2$
$\rightarrow$ **LMode** $p\ \ n_1$, **LMode** $p\ \ n_2$, **Página-Marco** $pf_2\ \ p\ \ n_1$,
Página-Marco $pf_2\ \ p\ \ n_2$

R_3 : **DetecEsc** $p\ \ n_1$, **LEMode** $p\ \ n_2$, **Página-Marco** $pf_2\ \ p\ \ n_2$
$\rightarrow$ **LEMode** $p\ \ n_1$, **Página-Marco** $pf_2\ \ p\ \ n_1$

R_4 : **DetecEsc** $p\ \ n_1$, $\neg$**LMode** $p\ \ n_1$, **LMode** $p\ \ n_2$,
Página-Marco $pf_2\ \ p\ \ n_2$, **Ok** $p \rightarrow$ **FaseInvalidación** $pf_2\ \ p\ \ n_1$

R_5 : **FaseInvalidación** $pf_1\ \ p\ \ n_1$, **LMode** $p\ \ n_2$,
Página-Marco $pf_1\ \ p\ \ n_2$, $\rightarrow$ **FaseInvalidación** $pf_1\ \ p\ \ n_1$

R_6 : **FaseInvalidación** $pf_1\ \ p\ \ n_1$, $\neg$**LMode** $p\ \ n_2$
$\rightarrow$ **LEMode** $p\ \ n_1$, **Página-Marco** $pf_1\ \ p\ \ n_1$, **Ok** p

R_7 : **DetecEsc** $p\ \ n_1$, **LMode** $p\ \ n_1$, **Página-Marco** $pf_1\ \ p\ \ n_1$, **Ok** p
$\rightarrow$ **FaseInvalidación** $pf_1\ \ p\ \ n_1$

Las relaciones contenidas en el multiconjunto caracterizan las entidades físicas (tabla de página, contenido de la página) y las entidades

lógicas (fase del protocolo por ejemplo). **LEMode** p n establece que la página p en el nodo n está en modo de lectura/escritura. **LMode** p n establece el derecho de lectura. **DetecLec** p n (análogamente para **DetecEsc** p n) establece que una excepción de lectura (o escritura) ha ocurrido en la página p en el nodo n. **Página-Marco** pf p n establece que una página marco pf (una página física) correspondiente a la página virtual p existe en el nodo n. **FaseInvalidación** pf p n establece que la página p está en fase de invalidación para que la página marco pf sea accesible en modos de lectura/escritura en el nodo n. **Ok** p establece que la página p no está envuelta en alguna fase de invalidación.

Cada regla expresa un cambio de estado con la precondición asociada. Por ejemplo, la ocurrencia de **LEMode** p n en el lado izquierdo pero no el lado derecho (como en R_2) implica que el derecho de lectura/escritura está cancelado para la página p del nodo n. Las reglas R_1 y R_2 corresponden al caso donde el nodo n_1 pide derecho de lectura para una página y ésta está siendo usada en ese momento para lectura (R_1) o para escritura (R_2) por otro nodo n_2. Las reglas R_3, R_4 y R_7 tratan el caso donde el nodo n_1 escribe sobre una página y ésta es accesada en modo de escritura (R_3) o en modo de lectura (R_4) por otro nodo n_2 o accesada por ella misma (nodo n_1) en modo de lectura (R_7). Finalmente, las reglas R_5 y R_6 tratan el ciclo de invalidación de copias de páginas y el derecho de lectura/escritura cuando todas las invalidaciones fueron hechas respectivamente.

6.4.1. Las propiedades

Una vez entendido el protocolo se debe encontrar alguna forma de verificar que el protocolo cumple el trabajo para el que fue pensado. Esto se puede hacer dando propiedades que debe cumplir el protocolo, buscando tener un invariante.

Un invariante está compuesto por propiedades P las cuales son defi-

nidas de acuerdo a la siguiente gramática:

$$\begin{aligned} P &::= \overline{\mathbf{R}} y_1 \ldots y_n \leq K \mid \overline{\mathbf{R}} y_1 \ldots y_n > K \\ &\quad \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \mid \neg P \mid \mathit{verdadero} \mid \mathit{falso} \\ K &::= 0 \mid 1 \end{aligned}$$

La notación $\overline{\mathbf{R}} y_1 \ldots y_n$ expresa la cardinalidad de una relación llamada R en el multiconjunto de acuerdo al patrón $y_1 \ldots y_n$ (donde y_i puede ser un nombre de variable o un comodín * para cualquier símbolo). Por ejemplo, en el multiconjunto $\{\!|\mathbf{R}\ a\ b, \mathbf{R}\ a\ c|\!\}$ tenemos que $\overline{\mathbf{R}}\ i * = 2, \overline{\mathbf{R}}\ i\ b = 1, \overline{\mathbf{R}}\ * * = 2, \overline{\mathbf{R}}\ i\ i = 0$. Las propiedades están implícitamente cuantificadas universalmente sobre sus variables libres.

A continuación mostramos los invariantes para el protocolo de Li y Hudak.

$$\begin{aligned} P_1 &: \overline{\mathbf{LEMode}}\ p * \leq 1 \\ P_2 &: \overline{\mathbf{LMode}}\ p * > 0 \Rightarrow \mathbf{LEMode}\ p * \leq 0 \\ P_3 &: \overline{\mathbf{LEMode}}\ p * > 0 \Rightarrow \mathbf{LMode}\ p * \leq 0 \\ P_4 &: \overline{\mathbf{DetecEsc}}\ p\ m > 0 \Rightarrow \mathbf{DetecLec}\ p\ m \leq 0 \\ P_5 &: \overline{\mathbf{DetecLec}}\ p\ m > 0 \Rightarrow \mathbf{DetecEsc}\ p\ m \leq 0 \\ P_6 &: \overline{\mathbf{FaseInvalidacion}}\ *\ p\ m > 0 \Rightarrow \mathbf{LEMode}\ p\ m \leq 0 \\ P_7 &: \overline{\mathbf{LMode}}\ p\ m > 0 \Rightarrow \\ &\quad (\mathbf{Página\text{-}Marco}\ pf\ p\ m > 0 \wedge \mathbf{Página\text{-}Marco}\ pf\ p\ m \leq 1) \\ P_8 &: \overline{\mathbf{LEMode}}\ p\ m > 0 \Rightarrow \\ &\quad (\mathbf{Página\text{-}Marco}\ pf\ p\ m > 0 \wedge \mathbf{Página\text{-}Marco}\ *\ p * \leq 1) \end{aligned}$$

La relación como $\overline{\mathbf{LEMode}}\ p *$ expresa el número de ocurrencias de las tuplas que satisfacen la relación que casan con los argumentos (* casa con cualquier valor). La propiedad $\overline{\mathbf{LEMode}}\ p * \leq 1$ establece que para una página p, hay a lo más un nodo m tal que $\mathbf{LEMode}\ p\ m$ existe. En el protocolo, es equivalente a decir que siempre a lo más un nodo puede escribir sobre una página p. La propiedad P_1 establece que a lo más un nodo puede tener acceso de lectura/escritura sobre una página. P_2 establece que si un nodo tiene acceso de lectura sobre una página, ningún otro nodo puede tener acceso de escritura sobre esa

página. P_3 establece el caso dual. P_4 y P_5 revisan que las fallas de lectura y escritura de una página no pueden ser detectadas al mismo tiempo en una página. P_6 establece que en la fase de invalidación, el nodo que dispara la invalidación no debería tener acceso de lectura/escritura sobre la página. P_7 establece que si hay un acceso de lectura a la página entonces este nodo no debería tener una y solo una copia física de la página. Finalmente P_8 verifica que si un nodo tiene acceso de escritura sobre una página, ésta debe tener una copia física de la página y no más de una copia de esta página debe existir en el sistema.

6.5. Algoritmo de verificación

Una vez formalizado el protocolo y las propiedades que se espera que cumpla, necesitamos alguna forma de verificar que las propiedades sean verdaderamente invariantes del protocolo. Esta verificación debe ser automática y relativamente eficiente para que se pueda aplicar a protocolos de tamaño real.

El algoritmo que se propone está basado en la simple observación de que una regla en un programa de Gamma quita e introduce un número finito y dado de relaciones. A partir de una propiedad que se satisfaga *después* de la aplicación de una regla, es posible derivar una propiedad que se debe satisfacer *antes* de que esta regla sea aplicada, es decir, una precondición. Por ejemplo, consideremos la propiedad P_2:

$$\overline{\mathbf{EMode}}\ p * \leq 0 \vee \overline{\mathbf{LEMode}}\ p * \leq 0$$

y la regla R_2:

$$\begin{aligned} R_2 \ : \ & \mathbf{DetecLec}\ p \quad n_1, \mathbf{LEMode}\ p \quad n_2, \mathbf{Página\text{-}Marco}\ pf_2 \quad p \quad n_2 \\ & \rightarrow \mathbf{LMode}\ p \quad n_1, \mathbf{LMode}\ p \quad n_2, \mathbf{Página\text{-}Marco}\ pf_2 \quad p \quad n_1, \\ & \mathbf{Página\text{-}Marco}\ pf_2 \quad p \quad n_2 \end{aligned}$$

El traslape entre P_2 y R_2 se muestra en la proyección (de las variables de regla R_2 a las variables de la propiedad P_2) μ:

$$\mu(p) = p \quad \mu(pf) = pf \quad \mu(n_1) = * \quad \mu(n_2) = *$$

En la regla R_2, una relación **DetecLec** y una relación **LEMode** se eliminan del multiconjunto. Al mismo tiempo, dos relaciones **EMode** y una relación **Página-Marco** también son agregadas. Considerando la cardinalidad de las variaciones, es decir, el cambio en la cardinalidad entre el lado izquierdo y derecho de la regla R_2, tenemos:

$$\begin{aligned} \Delta\overline{\textbf{DetecLec}}\ p* &= -1 \\ \Delta\overline{\textbf{LMode}}\ p* &= +2 \\ \Delta\overline{\textbf{LEMode}}\ p* &= -1 \\ \Delta\overline{\textbf{Página-Marco}}\ pf\ p* &= +1 \end{aligned}$$

Entonces, si P_2 debe ser verificada una vez que R_2 es aplicada, antes debemos asegurarnos de que:

$$\overline{\textbf{LMode}}\ p* \leq \underbrace{-2}_{-\Delta\overline{\textbf{LMode}}\ p*}$$
$$\vee\ \overline{\textbf{LEMode}}\ p* \leq \underbrace{1}_{-\Delta\overline{\textbf{LEMode}}\ p*}$$

Pero notemos que la propiedad $\overline{\textbf{LMode}}\ p* \leq -2$ es equivalente a FALSO ya que las cardinalidades no pueden ser negativas. Entonces la propiedad antes mencionada se reduce a $\overline{\textbf{LEMode}}\ p* \leq 1$. Además esta condición se debe satisfacer cuando R_2 es aplicable, es decir, cuando $\overline{\textbf{DetecLec}}\ p* > 0 \wedge \overline{\textbf{LEMode}}\ p* > 0 \wedge \overline{\textbf{Página-Marco}}\ pf\ p* > 0$. De esta forma obtenemos la *precondición más débil* de P_2 para la regla R_2:

$$\begin{aligned} &(\overline{\textbf{DetecLec}}\ p* > 0 \wedge \overline{\textbf{LEMode}}\ p* > 0 \\ &\wedge\overline{\textbf{Página-Marco}}\ pf\ p* > 0) \Rightarrow \overline{\textbf{LEMode}}\ p* \leq 1 \end{aligned}$$

Esta nueva propiedad se convierte en un invariante del protocolo. Entonces, la verificación del algoritmo está definido como un proceso que va construyendo iterativamente un invariante reforzado. Este proceso debe converger ya que el lenguaje de las propiedades es finito. El resultado es o bien FALSO, es decir, en caso de que no se puedan tener dichas condiciones, o una colección de propiedades (el invariante

reforzado) que se deben satisfacer por el estado inicial del sistema. La verificación de que esta condición se satisface también puede hacerse mecánicamente utilizando la técnica presentada.

El algoritmo de verificación se muestra a continuación:

Verifica(P, Prt) =
  TODO := {P}, DONE := $\emptyset$
  Prt := $\{Rl_1, \ldots, Rl_n\}$
  $PROY_i := \{\mu^i_1, \ldots, \mu^i_{k_i}\}$
    con $i \in [1, n]$ y donde $PROY_i$ es el conjunto
    de proyecciones de la regla Rl_i a P.
  while TODO $\neq \emptyset \wedge \mathcal{N}$otfalse(DONE $\cup$ TODO) **do**
    $Q \in TODO$
    TODO := TODO - {Q}
    DONE := DONE $\cup$ {Q}
    for $i \in [1, n], j \in [1, k_i]$ **do**
      $Q^i_j := \mathcal{N}(PMD(Q, Rl_i, \mu^i_j))$
      $R_1 \wedge \cdots \wedge R_m := Q^i_j$
      for $k \in [1, m]$ **do**
        if $\mathcal{N}ew(R_k$, DONE $\cup$ TODO) **then**
          TODO := TODO $\cup\{R_k\}$
        end if
      end for
    end for
  end while
  $\{P_1, \ldots, P_n\}$:= DONE
  resultado := $\mathcal{N}_1(P_1 \wedge \cdots \wedge P_n)$

El algoritmo empieza poniendo en TODO el conjunto de propiedades. Cada propiedad Q de este conjunto se considera que converge (lo cual ocurre cuando las nuevas propiedades son implicadas por las propiedades antes generadas). Luego se calcula el conjunto de proyecciones μ^i_j de las reglas Rl_i (del protocolo Prt) a la propiedad Q. Entonces, se hace la evaluación de la condición más débil Q^i_j de la propiedad Q para la regla Rl_i con proyección μ^i_j usando la función PMD. Una vez que esta nueva propiedad está normalizada a través de $\mathcal{N}$, (es decir, se

propagan las negaciones, se pone la proposición en forma normal conjuntiva, es decir, se reduce la proposición, etc) se agrega al conjunto TODO si esta propiedad no es implicada por las propiedades en TODO ∪ DONE (por medio de la llamada a la función $\mathcal{N}ew$). En cada paso de la iteración, se revisa si hay una contradicción en el sistema generado por los conjuntos TODO y DONE, llamando a la función $\mathcal{N}otfalse$.

Este algoritmo termina ya que el conjunto de propiedades que puede ser producido es finito ya que la cardinalidad del conjunto de propiedades también lo es, igual que la cardinalidad del conjunto de relaciones, así que el proceso de obtener las precondiciones más débiles también es un proceso finito. Se supone que los conjuntos de variables usadas para definir los protocolos y las propiedades (pues sin pérdida de generalidad podemos hacer un renombramiento).

Finalmente tenemos construido un conjunto de proposiciones que forman un invariante, lo único que resta por hacer es la verificación de que las nuevas propiedades se cumplen en el estado inicial del sistema.

6.5.1. Las funciones auxiliares

La precondición más débil. Una proyección de una regla Rl sobre una propiedad P es una función $\mu : FV(Rl) \cup \{*\} \rightarrow FV(P) \cup \{*\}$ tal que:

- $x, y \in VAR$,
  $x \neq y \Rightarrow \mu(x) \neq \mu(y) \wedge \mu(x) = \mu(y) = *$
- $\mu(*) = *$

Y está definida como sigue: $PMD(P, T_1\ T_2 \rightarrow T_3, \mu) =$

$$\underbrace{\bigwedge \overline{R_i} y_1 \dots y_n > K}_{\text{parte 1}} \qquad \begin{cases} K = 0|1, \forall R_i \text{ tal que} \\ Card(\{\mathbf{R}_i\ x_1 \dots x_n \in T_1| \\ \mu(x_j) = y_j \vee y_j = *\}) > K \end{cases}$$

$$\underbrace{\bigwedge \overline{R_i'} y_1 \dots y_n \leq 0}_{\text{parte 2}} \qquad \begin{cases} \forall R_i' \text{ tal que } \mathbf{R'}_i\ x_1 \dots x_n \in T_2, \\ \mu(x_j) = y_j,\ x_j \neq * \Rightarrow y_j \neq * \end{cases}$$

$$\Rightarrow \underbrace{\mathcal{S}\left(P\left[\frac{\overline{R_i''} y_1 \dots y_n + I_i}{\overline{R_i''} y_1 \dots y_n}\right]\right)}_{\text{parte 3}} \qquad \begin{cases} I_i = & \forall R_i'' \text{ que aparezca en } P \text{ con} \\ & Card(\{\mathbf{R"}_i\ x_1 \dots x_n \in T_3| \\ & \mu(x_j) = y_j \vee y_j = *\}) \\ & -Card(\{\mathbf{R"}_i\ x_1 \dots x_n \in T_1| \\ & \mu(x_j) = y_j \vee y_j = *\}) \end{cases}$$

El resultado de PMD toma la forma: parte 1 $\wedge$ parte 2 $\Rightarrow$ parte 3. La parte 1 y la parte 2 representan la condición de aplicación de la proposición actualizada P en la parte 3. La parte 1 y la parte 2 expresan el hecho de que la precondición calculada tiene que ser satisfecha cuando la regla es aplicable, es decir, cuando las relaciones de T_1 están en el multiconjunto (con cardinalidad estrictamente mayor que 0) y las relaciones de T_2 no están disponibles en el multiconjunto (con cardinalidad menor o igual que cero). La parte 3 se deriva de la proposición inicial P tomando en cuenta las modificaciones a las cardinalidades (de acuerdo a la creación y eliminación de las relaciones como lo indica la regla). Después, esta parte es simplificada por $\mathcal{S}$ que es una función que sirve para asegurarse de que la nueva propiedad pertenezca al lenguaje de las propiedades.

Función de normalización. La función $\mathcal{N}$ es empleada para normalizar cada propiedad antes de que sea pasada a la función $\mathcal{N}ew$. La tarea de la función de normalización es muy simple, quitar las negaciones propagándolas dentro de las expresiones, poner la proposición en forma normal conjuntiva, además, quitar las proposiciones redundantes y las ocurrencias de *verdadero* y *falso* reduciendo de esta

manera la proposición.

Detección de precondiciones nuevas. La función $\mathcal{N}ew$ verifica que una propiedad dada P no sea implicada por el conjunto de propiedades E:

$$\begin{aligned} \mathcal{N}ew(P, E) \;=\; & \text{sea } E = \{P_1, \ldots, P_n\} \\ & (P_1 \nRightarrow P) \wedge \cdots \wedge (P_n \nRightarrow P) \end{aligned}$$

6.6. Aplicación del protocolo

En el artículo de Le Métayer y Mentré [D. Le Métayer, 1997] se menciona una aplicación del protocolo de Li y Hudak que consiste en una implementación del algoritmo en Objective Caml 2, que es un lenguaje parecido a ML. Este programa usa bases de datos distribuidas para implementar la revisión $\mathcal{N}$otfalse y verificar las propiedades producidas contra las condiciones iniciales.

Ahora veamos como podemos llevar el protocolo propuesto a un SRC de la manera en que ya hemos explicado.

6.7. De Gamma a SRC

Dado que el protocolo lo podemos ver como un SRC, donde el conjunto de términos para el SRC sería el conjunto de multiconjuntos que manejamos en el protocolo, y las relaciones del SRC las vemos como fue propuesto en capítulos anteriores, la relación de un SRC será el programa mismo de Gamma. De esta forma podemos ver el protocolo como un SRC y pasamos ahora a demostrar que las propiedades propuestas son un invariante para el protocolo.

Una vez que verificamos que las nuevas propiedades construidas a través del algoritmo antes de que se aplique el protocolo (es decir, en el estado inicial del sistema mediante un algoritmo que vaya recorriendo cada propiedad y se vaya verificando), podemos continuar con la verificación de que el protocolo satisface las propiedades que forman el invariante. Esto lo llevamos a cabo mediante la lógica propuesta en

el capítulo sobre los SRC, comprobando de esta manera que cada uno de los estados por los que va pasando el sistema en cada aplicación de las reglas, es decir, por cada multiconjunto o término que se obtiene al aplicar las relaciones de reescritura del sistema durante la ejecución del protocolo cumple las condiciones que forman el invariante.

Esto lo llevaremos a cabo empleando la lógica (propuesta en el capítulo 3.4) y la semántica para SRC (§3.3.2), mediante la aplicación de las reglas (mediador, terminación, etc), como se muestra a continuación.

6.7.1. Aplicación de la lógica para los términos de un SRC

Como mencionamos anteriormente, la forma en que mostraremos que las propiedades del invariante se cumplen durante la ejecución del protocolo (SRC) es mediante la aplicación de las reglas de inferencia propuestas.

De esta forma estaríamos verificando que las propiedades del invariante propuesto se cumplen en cada término (o estado que describe al sistema de MVC) y de esta forma el protocolo (o SRC) sería correcto.

Entonces una vez que tenemos la lógica para hacer las demostraciones que nos interesan, veamos como podemos escribir lo que queremos probar de acuerdo al lenguaje lógico sobre términos que propusimos.

Queremos verificar que cada multiconjunto o término por los que pasa el sistema, es decir, las trazas que genera el protocolo, cumplen con una condición dada del invariante.

Lo anterior lo podemos ver de la siguiente manera:

$$\forall \alpha . P \models \diamond \bar{\alpha} \text{ y } \forall M_k \text{ que aparece en } \bar{\alpha}, \quad M_k \models Inv$$

donde P representa el protocolo (o SRC), $\diamond \bar{\alpha}$ representa a las trazas producidas, M_k es el multiconjunto (o término del SRC) correspondiente a una regla, y finalmente Inv es el invariante (o una propiedad sobre los términos, expresada en $L(T)$).

Entonces, la hipótesis inductiva si pensamos en una demostración

sobre las trazas producidas sería:

$$P \models \diamond\bar{\alpha} \text{ y } \forall M_k \in \bar{\alpha}, M_k \models Inv$$

Ahora, lo que queremos demostrar es:

$$M \rightarrow_P M' \quad \wedge \quad M \models Inv \quad \wedge \quad M' \models Inv$$

$$\Longrightarrow$$

$$P \models \diamond(M, M') \odot \bar{\alpha}$$

Es decir, sabemos por hipótesis de inducción, que el protocolo hasta cierto punto, produce una traza $\bar{\alpha}$ tal que todo término M_k que la compone cumple con el invariante Inv. Lo que queremos ver ahora es que una traza mayor también cumple el invariante, es decir, queremos demostrar ahora que la nueva traza producida por el protocolo cumple también el invariante, pero eso se reduce a verificar que el protocolo puede producir esta nueva traza (el consecuente de la implicación anterior) pues la traza anterior $\bar{\alpha}$ ya cumplía con el invariante (hipótesis de inducción) y la nueva pareja de términos (M, M') también (por la implicación anterior, asumiendo el antecedente).

Usando las reglas antes mencionadas para la semántica y lógica propuestas:

$$\frac{t \models_T \psi_S, \quad t' \in RT(R,t), \quad S \models_{SRC} \diamond\bar{\alpha}}{S \models_{SRC} \diamond(t,t') \odot \bar{\alpha}} \; Mediador \qquad \frac{t \not\models \psi_S}{S \models_{SRC} \diamond(t,t)} \; Terminal$$

donde $t, t' \in T, \psi \in L(T)$ y $\bar{\alpha} \in \mathcal{T}(T)$.

Es claro que mediante la aplicación de esta regla podemos tener una demostración por inducción sobre las trazas que generan las aplicaciones de las reglas que forman el protocolo, mostrando así que cada multiconjunto por el que va pasando el sistema cumple con el invariante.

Debido a que la construcción del invariante Inv para el protocolo en cuestión está dada por un algoritmo, no tenemos completo el invariante hasta que este algoritmo ha terminado. Por cuestiones prácticas, aquí no hemos dado explícitamente el invariante completo, sino que hemos mostrado que mediante el algoritmo dado es posible su elaboración. Por esta razón, la verificación de que la propuesta de protocolo realmente lo es, tampoco es dada explícitamente sino que se demuestra que el invariante se cumple durante la ejecución del protocolo, utilizando la regla de la lógica dada anteriormente.

Es decir, para una regla dada y una propiedad del protocolo, podemos hacer la demostración de que el protocolo siempre la cumple: lo que tenemos que probar es que antes y después de aplicar la regla R_2:

$$\begin{aligned} R_2 \ : \ & \textbf{DetecLec } p \quad n_1, \textbf{LEMode } p \quad n_2, \textbf{Página-Marco } pf_2 \quad p \quad n_2 \\ & \rightarrow \textbf{LMode } p \quad n_1, \textbf{LMode } p \quad n_2, \textbf{Página-Marco } pf_2 \quad p \quad n_1, \\ & \textbf{Página-Marco } pf_2 \quad p \quad n_2 \end{aligned}$$

y tenemos la propiedad P_2 una vez que el algoritmo es aplicado, se cumple la propiedad correspondiente de la propuesta de invariante:

$$\begin{aligned} (\overline{\textbf{DetecLec}}\ p * > 0 \wedge \overline{\textbf{LEMode}}\ p * > 0 \\ \wedge \overline{\textbf{Página-Marco}}\ pf\ p * > 0) \Rightarrow \overline{\textbf{LEMode}}\ p * \leq 1 \end{aligned}$$

Sea M el multiconjunto o término que describe el estado antes de aplicar la regla R_2, es decir, M describe el estado de las páginas y los nodos del sistema.

Ahora veamos que un término T (o multiconjunto) después de aplicar la regla R_2 cumple la propiedad P_2 del invariante, esto es, tenemos que verificar que antes de la aplicación de R_2, el estado inicial del sistema es que tenemos una petición de lectura sobre una página p en un nodo n_1, pero ésta está siendo escrita por un nodo n_2. De esta forma, vemos que tenemos una petición de lectura, por lo tanto $\overline{\textbf{DetecLec}}\ p * > 0$, además tenemos a n_2 escribiendo sobre p, así que $\overline{\textbf{LEMode}}\ p * > 0$ y tenemos que tenemos una copia física para la página p, por lo tanto $\overline{\textbf{Página-Marco}}\ pf\ p * > 0$, ahora veamos que se cumple $\overline{\textbf{LEMode}}\ p * \leq 1$, pero esto es cierto, pues sabemos que tenemos solo a n_2 escribiendo sobre p. Por lo tanto tenemos que $M \models P_2$.

Ahora consideremos el estado (multiconjunto o término) M' que describe el estado del sistema después de haber aplicado la regla R_2 y veamos que $M' \models P_2$.

Es claro que se cumple la proposición P_2 en M' pues así fue como se construyeron las propiedades. Una vez que se aplica R_2 no hay nodos que hagan la petición de lectura, por lo tanto la propiedad P_2 es cierta. Entonces sabemos que $M' \models P_2$.

Sea $\bar{\alpha}$ una posible producción de trazas generada a partir del término M', de tal forma que tenemos $M' \models \bar{\alpha}$.

Ahora podemos aplicar la regla *Mediador* y tenemos que $P \models \diamond(M, M') \odot \bar{\alpha}$.

La verificación en las demás propiedades se hace de manera similar, de tal forma que demostramos que cada una de las propiedades del invariante se cumplen durante la ejecución del protocolo.

De esta forma la verificación de las condiciones para tener un invariantes es realizada mostrando que las propiedades deseadas en el protocolo se conservan durante su ejecución, y por tanto podemos afirmar que el protocolo es correcto.

Con esto terminamos la verificación de propiedades para una aplicación a través de la formalización propuesta en este trabajo.

Capítulo 7

Conclusiones

El presente trabajo de tesis surge como parte del proyecto de investigación "Semántica del paralelismo: en busca de una unificación" (véase §1.4). El trabajo de investigación de esta tesis se basa en un profundo estudio de los SRC y del lenguaje paralelo de programación Gamma así como de su semántica. A través de este estudio, se logran entender los conceptos fundamentales para llevar a cabo una propuesta de acuerdo a la hipótesis de partida: proponer una generalización del modelo semántico de Gamma que permitiera ser la base para un modelo semántico de los SRC y de este modo, de una subclase de lenguajes paralelos. Puntualizando la hipótesis, la tesis tiene los siguientes objetivos como parte de los objetivos del proyecto antes mencionado:

1. Extraer un modelo matemático más abstracto de la semántica de Gamma diseñada por [Hernández, 1999].
2. Caracterizar qué lenguajes de programación paralela pueden considerarse sistemas compuestos de reducción.
3. Formular un modelo semántico para todos los sistemas de reducción compuestos basado en la versión abstracta de la semántica de Gamma.

[1]. Para alcanzar los objetivos planteados, la tesis presenta un estudio de la semántica de Gamma.

[1] véanse los objetivos generales del proyecto de investigación en §1.4

El estudio de Gamma y los SRC consiste en entender la semántica de los lenguajes de programación para luego estudiar la semántica de los lenguajes paralelos como Gamma y de esta manera entender la semántica de los SRC. La tarea de comprender la semántica de los lenguajes paralelos es realizada mediante el estudio de un lenguaje de programación muy sencillo, revisando las semánticas operacional, denotacional y axiomática. Una vez que se comprendieron las formas en que se puede describir el comportamiento de un programa de un lenguaje secuencial, se extiende este lenguaje para comprender el paralelismo y de esta forma estudiar la semántica de un lenguaje paralelo. Pasamos después a revisar los SRC: primero se estudia el comportamiento de los sistemas de reducción y se dan algunos ejemplos de este modelo de cómputo (como el cálculo lambda). Una vez que la idea de los sistemas de reducción ha quedado clara, se presentan los operadores de composición paralela y secuencial para los SRC. Se estudian también algunas de las características de los SRC basadas en estos operadores de composición. Finalmente se presenta un estudio de la semántica de los SRC y se establece una lógica para los SRC. En este punto se tienen las herramientas para empezar con el desarrollo de la propuesta de esta tesis.

La propuesta desarrollada en el presente trabajo fue la de transformar los programas de Gamma y Gamma estructurado en SRC y de esta forma obtener una generalización de estos programas. Además de seguir con la propuesta inicial, se propone una semántica para los SRC[2] y una lógica para este modelo con el fin de que el modelo semántico propuesto permita desarrollar algunas técnicas de verificación y transformación de programas usando la lógica antes mencionada.

Para llevar a cabo la transformación de los programas de Gamma y Gamma estructurado a SRC se siguió la idea de tener una relación para dos multiconjuntos M, M' (o multiconjuntos estructurados según corresponda) del conjunto de multiconjuntos (o multiconjuntos estructurados) sobre los que actuaba el programa de Gamma (o Gamma

[2]basada en la propuesta del modelo semántico hecha por el Dr. Francisco Hernández Quiroz (véase [Hernández, 1999])

estructurado) P. La relación estaba definida como sigue: los multiconjuntos M, M' (o multiconjuntos estructurados) estaban en la relación si la aplicación de P sobre M nos llevaba a M'. Por la naturaleza de Gamma, podíamos componer los programas de manera secuencial y paralela y de esta forma obteníamos un SRC correspondiente al programa de Gamma (o Gamma estructurado) P, donde la relación de reescritura es la que describimos anteriormente y los términos eran los multiconjuntos sobre los que aplicábamos el programa P.

Un ejemplo de la transformación propuesta se estudia en el capítulo 6 donde se realiza la transformación a un caso real: un protocolo de coherencia para un sistema de memorias virtuales compartidas escrito en Gamma. Empleando la transformación de Gamma (o Gamma estructurado) en SRC que se propone en este trabajo, se obtiene un SRC que corresponde al protocolo original escrito en Gamma.

Otra de las razones para tener esta aplicación es la de llevar a cabo la verificación de algunas propiedades del protocolo. Esta verificación de propiedades nos permitiría probar la corrección del protocolo de manera formal. Esta tarea es realizada trabajando sobre el SRC resultante de aplicar la transformación propuesta, de tal forma que las propiedades que se quiere cumpla el protocolo, se expresan como una propuesta de invariante y se hace una demostración de que estas propiedades se cumplen antes, durante y después de la aplicación de las reglas de reescritura del SRC mostrando de esta forma que el protocolo de coherencia es correcto. Finalmente con este trabajo hecho sobre la aplicación, se completa la tarea de llevar a cabo la verificación de programas usando la transformación, semántica y lógicas propuestas para Gamma y los SRC de acuerdo a los objetivos del proyecto del cual surgió este trabajo.

La verificación de la corrección del protocolo se realiza empleando la semántica y la lógica propuestas para los SRC en el capítulo 3.[3] Con estas herramientas para los SRC, es posible hacer demostraciones sobre

[3]Las propuestas hechas sobre la semántica y la lógica para los SRC están basadas en la semántica y la lógica propuesta por el Dr. Francisco Hernández Q. haciendo una adaptación para los SRC (como se había mencionado anteriormente).

programas usando las trazas generadas por la aplicación de las reglas de reescritura del SRC sobre los términos (que para el protocolo en cuestión son multiconjuntos) y tener de esta manera una demostración formal de su corrección siguiendo la idea de las demostraciones por inducción sobre las trazas.

Además, con la transformación propuesta se logra que se establezcan bases para realizar transformaciones sobre los programas de otros lenguajes paralelos que tengan las características de Gamma, es decir, la subclase de lenguajes paralelos que puedan transformarse en SRC y de esta forma tener una generalización para esta subclase de lenguajes paralelos. Además, se tiene un sistema de verificación para los SRC en base a la semántica y lógica propuestas. Con este sistema de verificación y la prueba de que esta transformación se puede extender, podemos seguir la idea de tener un sistema de verificación para esta subclase de lenguajes paralelos.

De esta forma se cumplen los objetivos[4] iniciales planteados para la realización de este trabajo y se trazan algunas posibles rutas para continuar con las líneas de investigación que siguió esta tesis.

[4] véase las bases de la investigación en §1.4

Capítulo 8

Bibliografía

Bibliografía

[Abrahamsky and Gabbay, 1990] Abrahamsky, S. and Gabbay, D. M. (1990). *Handbook of Logic in Computer Science*, volume 2. Oxford Science Publications.

[Chaudron, 1998] Chaudron, M. R. V. (1998). *Separating Computational and Coordination in the Design of Parallel and Distributed Programs.* New arts, foundation.

[D. Le Métayer, 1997] D. Le Métayer, D. Mentré, T. P. (1997). Formalization and verification of coherence protocols with the gamma framework. *IRISA, France.*

[De La Cruz Martínez, 2002] De La Cruz Martínez, G. (2002). Semántica de lenguajes de orden superior. el caso de gamma.

[Hankin C., 1998] Hankin C., Le Métayer D., . S. D. (1998). refining multiset transformers". *Theorical Computer Science.*

[Hernández, 1999] Hernández, F. (1999). *A Semantics-Based proof system for Gamma.* Technology and medicine, Imperial College of Science.

[Li and Hudak, 1986] Li, K. and Hudak, P. R. (1986). Memory coherence in shared virtual memory systems. In *5th Annual ACM symposium on Principles of Distributed Computing*, pages 229–239. ACM.

[P. Fradet, 1998] P. Fradet, D. L. M. (1998). Structured gamma. *Science of Computer Programming*, (31):263–289.

[Sands, 1996] Sands, D. (1996). Composed reduction systems. *Science of Computer Programming.*

[Tarski, 1941] Tarski, A. (1941). On the calculus of relations. *The Journal of symbolic logic.*

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages an Introduction.* The MIT Press.