UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

COTAS DE COMPLEJIDAD EN LA RELACIÓN DE XML CON LOGICAS Y AUTÓMATAS

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS (COMPUTACIÓN)

P R E S E N T A:

JOSÉ CRUZ CARVAJAL CIPRÉS

DIRECTOR DE TESIS: DR. VLADISLAV KHARTCHENKO

México, D.F. **2009.**

A mi Director de Tésis: Dr. Vladislav Khartchenko por su paciencia algorítmica.

Al Dr. David Rosenblueth Laguette por su verificación incremental.

Al Dr. Ricardo Paramont Hernández García por su eclecticismo genético.

Al Dr. Angel López Gómez por su persistencia recursiva.

Al Dr. Juan Carlos Chimal Eguía por su guía auto-organizada.

Al Dr. Hugo Coyote Estrada por su motivación computacional.

Al Dr. Jaime Navarro Fuentes por su exigencia matemática.

Agradecimientos

A la Universidad Nacional Autónoma de México

Al Instituto de Investigación en Matemáticas Aplicadas y Sistemas

A la Facultad de Estudios Superiores Cuautitlán

A mi Tutor: Dr. Vladislav Khartchenko.

A mis maestros.

A la Dra. Suemi Rodríguez

Al Dr. Boris Escalante

Al personal de la Coordinación del IIMAS, en especial a: Lulú, Diana y Amalia

Índice general

1. INTRODUCCIÓN

En los cimientos de XML reconocemos la teoría de autómatas, la lógica matemática y la teoría de base de datos, el estudio de este lenguaje ha generado por más de una década una gran cantidad de imbricaciones entre estas tres áreas, tan inesperadas como elegantes. El estudio de las relaciones entre los autómatas y la lógica formal es tan antiguo como la misma teoría de la ciencia computacional; Turing en 1936 muestra como describir el comportamiento de su máquina universal con una fórmula expresada mediante un predicado de la lógica de primer orden, lo que le permite concluir la inexistencia de un algoritmo que decida la validez de las proposiciones en esta lógica.

XML significa lenguaje de marcas generalizado (eXtensible Markup Language). Es un lenguaje computacional de marcado genérico, usado para estructurar información en un documento o en general en cualquier archivo que contenga datos, como archivos de configuración de un programa o una tabla de datos. Constituye un estándar abierto y libre, creado por el World Wide Web Consortium (W3C), en colaboración con un panel que incluye representantes de las principales compañías productoras de software. XML fue propuesto en 1996, y la primera especificación se oficializó en 1998. Desde entonces su uso ha tenido un crecimiento acelerado, y representa la base para la construcción evolutiva de la web actual hacia la web semántica.

El W3C es promotor de estándares que facilitan el intercambio de informaciones en la Web, recomienda la sintáxis XML para expresar lenguajes de marcado especializados, por ejemplo: XHTML, SVG, XSLT, MathXL. Su objetivo inicial es facilitar el intercambio automatizado de contenido entre sistemas con información heterogénea (interoperabilidad). XML concebido como una simplificación del Lenguaje Estándar Generalizado de Marcado o SGML (Standard Generalized Markup Language), retiene los principios esenciales de éste.

La estructura anidada de los documentos XML puede representarse mediante un árbol etiquetado y sin anchura definida (del inglés unranked, que hemos traducido de manera indistinta sin anchura definida o sin rango fijo). Ésta característica permite su análisis a partir de los formalismos que la teoría de autómatas y la lógica computacional han probado para codificar, recorrer, validar, consultar y transformar estructuras arborescentes. En este trabajo de investigación estudiamos comparativamente las cotas de la complejidad al validar documentos XML, modelados estos como un árbol ordenado etiquetado y sin rango o anchura definida o LUOT (Labeled Unranked Ordered Tree), mediante el sistema de tipificado DTD (Document Type Definition). También analiza las cotas de la complejidad que encontramos al realizar consultas a un documento XML mediante expresiones del lenguaje XPath. Para ambos procesos el análisis se hace tanto desde la perspectiva de la complejidad exclusivamente de los datos (con un tipo fijo), como de la complejidad combinada (con datos y su tipificación como entrada).

Consideremos, por ejemplo, el caso de un organizador de un congreso científico que requiere guardar el programa del congreso en un documento XML para subirlo a la Web y que este disponible para su consulta. Una manera de construirlo puede ser la siguiente:

```
<congreso>
   <apartado>
     <sesion>
       <presidente> V. Vianu</presidente>
         <conferencia>
            <titulo>On the Power of Tree Automata</titulo>
            <exponente>F. Niven</exponente>
         </conferencia>
         <conferencia>
            <titulo>Semistructured Data</titulo>
            <autores>S. Abiteboul, D. Suciu</autores>
         </conferencia>
     </sesion>
     <descanso> Café </descanso>
     <sesion>
        . . .
     </sesion>
   </apartado>
   <apartado>
     . . .
   </apartado>
</congreso>
```

El árbol que representa el documento solamente refleja la estructura sin los datos contenidos, los requerimientos establecidos con la estructura correspondiente al evento son:

- El congreso se puede dividir en varios apartados.
- Cada apartado (o el mismo congreso si no esta dividido en apartados), se compone de sesiones, cada una formada de una o más conferencias.
- En toda sesión se tiene un presidente de sesión que conduce las conferencias y coordina la discusión.
- Una conferencia puede tener varios exponentes, en cuyo caso se especifican el título y los autores; o un solo exponente, especificando el título y al conferencista.
- Entre las sesiones se pueden programar descansos.

Estos requerimientos prefiguran un lenguaje de árbol sobre un alfabeto formado por las marcas tales como congreso, apartado, sesión, presidente, etc. contenidas en el documento. En este

modelo ignoramos los datos y únicamente tomamos en consideración a los elementos de la estructura, esta descripción en forma de lenguaje de árbol se le denomina *tipo* o *esquema*. Un documento XML se dice que es válido, respecto a un esquema, si el documento (representado como un árbol) pertenece al lenguaje de árbol definido por el esquema.

Podemos encontrar varios lenguajes, llamados lenguajes de esquema o tipificado XML, que permiten describir tales requerimientos de los documentos. Esto no debe considerarse como referencia a la sintaxis o a la semántica de estos lenguajes de esquema, sino con relación a los aspectos teóricos y su conexión con la teoría de autómatas, en particular se deja a un lado la representación real de los datos que implican los documentos, es decir los tipos de datos, ya que el enfoque de análisis se dirige solamente a la estructura. En particular el estudio se interesa por estos lenguajes, en aspectos tales como:

- El problema de pertenencia (membership) corresponde a verificar si un documento es válido respecto un esquema establecido.
- El problema de la membresía uniforme corresponde a la validación del documento respecto a un determinado esquema.
- El problema de tener un conjunto vació (emptiness) corresponde al aspecto de investigar si, para un esquema, existe algún árbol válido respecto a ese esquema.
- El problema de inclusión esta relacionado con el aspecto de si todos los documentos que son válidos para un esquema, también lo son respecto a otro esquema.

Una Definición de Tipo de Documento o DTD (Document Type Definition) es básicamente una gramática libre de contexto o CFG (Context-Free Grammar) con expresiones regulares en el lado derecho de las reglas de producción. El lenguaje de árbol definido por la DTD esta formado por todos los árboles de derivación de su gramática, como sabemos para una CFG el conjunto de árboles de derivación forman una lenguaje regular de árbol, estos conceptos los llevamos a conjuntos sin anchura definida o rango variable

Siguiendo con el ejemplo anterior, el DTD que formaliza los requerimientos para documentos que describen programas de un congreso sería el mostrado a continuación:

```
<!DOCTYPE CONGRESO [
<!ELEMENT congreso (apartado + | (sesion, descanso?)+)>
<!ELEMENT apartado (sesion, descanso?)+)>
<!ELEMENT sesion (presidente, conferencia*)>
<!ELEMENT conferencia ((titulo,autores)|(titulo),exponente))>
<!ELEMENT presidente (#PCDATA)>
<!ELEMENT descanso (#PCDATA)>
<!ELEMENT titulo (#PCDATA)>
]>
```

Los símbolos utilizados en las expresiones en el lado derecho de las reglas se interpretan como sigue:

```
| = selección, ,= secuencia, + = una o más veces, ?= una o ninguna vez
```

Todos los elementos omitidos en las declaraciones (por ejemplo autores, exponente, etc.) se asume que son #PCDATA o *dato con caracter de particionado*, los cuales denotan simplemente secuencias de caracteres arbitrarios que codifican los datos dentro del documento.

La gramática que corresponde al DTD es como sigue:

congreso	$\rightarrow$	$apartado^+ + (sesion(descanso + \epsilon))^+$
apartado	$\rightarrow$	$(sesion(descanso + \epsilon))^+$
sesion	$\rightarrow$	$presidente \quad conferencia^+$
conferencia	$\rightarrow$	$(titulo \quad autores) + (titulo \quad expositor)$
presidente	$\rightarrow$	$DATA$
descanso	$\rightarrow$	$DATA$
titulo	$\rightarrow$	$DATA$

En la terminología de XML, al lado derecho de una regla se le da el nombre de *modelo de contenido* (content model). En estas condiciones podemos definir a un DTD como la tripleta:

$$D \quad = \quad (\Sigma, \quad s, \quad \delta)$$

con um símbolo inicial $s \in \Sigma$, y un mapeo δ que relaciona a cada símbolo del alfabeto Σ a una expresión regular formada a partir de Σ. La forma más fácil de definir el lenguaje generado por una DTD es interpretarla como un autómata de árbol tipo seto (hedge automaton) con el conjunto de estados correspondiendo al alfabeto, sea:

$$\mathcal{A}_D = (Q_D, \Sigma, Q_{D_f}, \Delta_D)$$

definido por: $Q = \Sigma$, $Q_{D_f} = \{s\}$, y:

$$\Delta_D = \{a(\delta(a)) \rightarrow a | a \in \Sigma\}.$$

$\mathcal{A}_D$ es un autómata tipo seto finito determinista o DFHA (Deterministic Finite Hedge Automaton), podemos definir al lenguaje $L(D)$ definido por D para que corresponda al lenguaje $L(\mathcal{A}_D)$.

Los lenguajes definidos por una DTD reciben la denominación de lenguajes locales, debido a que para un árbol t, la pertenencia de t a $L(D)$ se puede decidir observando todos los subpatrones de altura 1.

No es difícil ver que estos lenguajes locales forman una subclase propia de los lenguajes reconocibles, todo lenguaje finito que contiene los dos árboles:

$$\text{a} \quad \text{(b} \quad \text{(} \quad \text{c} \quad \text{d))} \quad \text{y} \quad \text{(a'} \quad \text{(} \quad \text{b} \quad \text{(} \quad \text{d} \quad \text{c)))}$$

no puede definirse por una DTD ya que para toda DTD, D tal que $L(D)$ contiene estos dos árboles también contiene al árbol:

$$a \quad (\quad b \quad (\quad c \quad d \quad) \quad)$$

La teoría descriptiva de la Complejidad Computacional enfoca su estudio desde la perspectiva de la Lógica, es decir caracteriza mediante lógicas las clases de complejidad computacionales, tradicionalmente definidas en términos de máquinas de Turing con recursos físicos (espacio y tiempo) acotados. Fagin en 1974 demostró que la clase de complejidad NP corresponde al conjunto de problemas que pueden ser descritos mediante la lógica de segundo orden (SO), es decir que la complejidad computacional de un problema dado puede ser representada por la riqueza expresiva del lenguaje que se requiere para especificar ese problema. La complejidad descriptiva tiene una participación fundamental en la teoría de base de datos, ya que una de tipo relacional es una estructura lógica finita y los lenguajes de consulta, que se usan con estas bases de datos, bien estructurados en forma tabular, corresponden a extensiones de la lógica de primer orden (FO).

Un documento XML constituye una base, o depósito, de datos semiestructurados. El aporte de la teoría de la complejidad descriptiva consiste en poder darle una estructura matemática, en lugar de una meramente de ingeniería (grado de utilización de los recursos tiempo y espacio), a las relaciones de un lenguaje, en este caso XML, con la lógica y los autómatas que impone su utilización bajo la perspectiva de su manipulación como una base de datos semiestructurados, de forma arborescente sin anchura definida y con nodos etiquetados.

El interés de esta investigación radica en determinar la complejidad computacional a través de los formalismos de procesamiento (es decir de un autómata) y a través de los formalismos lógicos (es decir un lenguaje lógico) al validar y consultar documentos XML, es decir una base con datos semi-estructurados en árboles sin anchura definida y nodos etiquetados, expresando las consultas en un lenguaje lógico, que se convertirá en el autómata que compute la consulta teniendo como entrada al documento XML.

A la codificación de un documento XML se le considera bajo los dos enfoques recomendados por el 3WC, la tipo cadena estructurada utilizada por DOM (Document Object Model) y la de SAX (Simple API for XML) que consiste en una cadena representando una estructura de apuntadores. Este trabajo no cubre direcciones de investigación, todavía considerados como problemas abiertos en este campo, de la codificación de documentos XML a una forma de flujo de datos (streaming).

La primera familia de problemas abordado en este estudio es la verificación de tipos en documentos XML modelados como árboles etiquetados. Se considera el sistema de tipificación DTD, con dos variantes en la evaluación de la complejidad de la validación, en el primero de ellos, el tipo es fijo y la entrada consiste solamente en el árbol de datos, en el segundo el tipo también es parte de la entrada. Esta metodologia es usual en la Teoria de Modelos Finitos, al primer caso se le llama la complejidad de los datos para el problema y mide la complejidad como función exclusivamente del tamaño de los datos; al segundo caso se le denomina complejidad combinada y da una medida más precisa de la dificultad del problema, ya que incluye como entrada, junto con los datos, a la tipificación.

La misma metodología descrita se utiliza para evaluar la complejidad teórica de la consul-

ta de documentos XML, mediante el lenguaje de consulta XPath. Algunos de los algoritmos analizados para este proceso reduce un árbol con rango no definido a un caso de un árbol con rango fijo, esto se efectúa durante la codificación de la estructura arborecente que modela al documento. La utilización del marco teórico de la Complejidad Descriptiva implica que limitamos nuestro estudio a procesos computacionales de tipo finito, es decir todos los objetos relevantes de nuestro universo se deben modelar mediante estructuras lógicas no infinitas. Esta limitación por otro lado nos ofrece la utilización de los sólidos fundamentos de formalización y las posibilidades combinatorias que nos ofrece la Teoría de Modelos Finitos.

Otra dirección de investigación que queremos consignar, al considerar los fundamentos de XML, tiene que ver con encontrar las lógicas, que son equivalentes a aquellas identificadas de forma directa, (FO y MSO), cuya evaluación a nivel de verificación de modelos es polinomial al tamaño de la fórmula y al del árbol. Es conocido que tanto la MSO como la FO tienen una complejidad inconveniente en el caso de verificación de modelos de estructuras arborecentes. La investigación en esta dirección requiere se incluya algún tipo de lógica modal, por ejemplo, sabemos que la CTL con operador en tiempo pasado, amplia y exitósamente aplicada por la comunidad de investigadores en la verificación de modelos (model checking), es equivalente a la FO en árboles sin rango fijo.

2. XML = AUTÓMATAS + LÓGICA + BASE DE DATOS

2.1. Conceptos Básicos

XML es un lenguaje diseñado para estructurar información en un documento o en general en cualquier archivo que contenga texto. Ha concitado una creciente aceptación en los últimos años por su caracter de estándar abierto y libre, creado por el Consorcio World Wide Web, W3C, en colaboración con un panel que incluye representantes de las principales compañías productoras de software. XML fue propuesto en 1996, y la primera especificación apareció en 1998. Desde entonces su uso ha tenido un crecimiento acelerado, con tendencia a continuar durante los próximos años; hoy en día tenemos la impresión que de repente todo el mundo lo está usando, o necesita usar este lenguaje. Una aplicación primaria de XML es su uso como estándar en el intercambio de información, donde esta posicionando ya como *lingua franca* en este rubro.

Un documento XML puede modelarse, con base en sus reglas sintácticas, como una estructura arborescente sin rango constante (unranked) con nodos etiquetados y ordenados, lo que permite analizarlo a través de numerosas formalizaciones ya estudiadas por la teoría de autómatas de árbol y de la lógica. XML derivado a partir del SGML, como el pionero HyperText Markup Language (HTML) para la presentación de documentos en los sitios Web, se diferencia de este, aparte de su enfoque casi exclusivo hacia la presentación de documentos del HTML, en que las etiquetas (por ejemplo, $< tag > \ldots < /tag >$) dentro de un documento XML son definidas por el mismo usuario del lenguaje, y que su interés se dirige a la descripción de los datos y a lo que representan. Cuando estudiamos los fundamentos de XML, solamente nos interesa la estructura del árbol, por consiguiente omitimos el texto ubicado entre las etiquetas. Por definición, tenemos un árbol diferente cuando se intercambia el orden de los descendientes, también es de hacerse notar que dada su característica de no tener una anchura definida (unranked), un nodo en un árbol XML puede tener una cantidad arbitrariamente grande de hijos.

En este trabajo de investigación estamos interesados en las clases de la complejidad alcanzadas tanto por el formalismo del procesamiento del lenguaje (es decir el autómata) como el del formalismo de la lógica empleada, ambas aplicadas al tratamiento de arboles sin anchura definida. Su intención, es pues, examinar el desempeño de las operaciones de navegación, de validación y de consulta de documentos XML en forma de un lenguaje lógico, lo cual nos posibilita convertirlo a un autómata cuyo funcionamiento recorre, valida o consulta un documento XML. Al trabajar con documentos XML, en primer instancia nos interesa distinguir los documentos que tienen sentido de aquellos que no lo tienen, cuando dos entidades quieren

intercambiar información utilizando XML requieren como base un formato que ambas partes aceptan. Otro requerimiento elemental consiste en recorrer un árbol para buscar a través de todos los nodos de un documento aquellos que cumplan con una serie de condiciones en la estructura de sus etiquetas. La intransigencia de XML con la estructura que debe tener un tipo determinado de documentos, hace factible conservar cotas de complejidad que posibilita implementar algoritmos que extraigan automáticamente información de un conjunto de documentos, por ejemplo para crear y manipular bases de datos o listados con información sobre ellos.

Antes de ser lanzado XML, ya existían otros lenguajes de marcas, como por ejemplo el HTML, también basado en SGML. El problema de SGML, debido a su alto grado de generalidad y flexibilidad, el análisis sintáctico de un documento se vuelve complejo y difícil. XML es más estricto su sintaxis que SGML, lo que hace más fácil la construcción de bibliotecas de programas para procesarlo. Comparado con otros sistemas usados en la creación de documentos, el XML tiene la ventaja de poder ser mas exigente en cuanto a su organización, lo cual resulta en documentos mejor estructurados. Por ejemplo en LaTeXexisten también marcas que permiten estructurar un documento, al identificar el nombre del autor y el título del documento mediante los comandos *{author}* y *{title}* sin embargo no exhige a los redactores de documentos a que usen estas marcas y algunos de ellos pueden introducir el título de forma que aparezca visualmente igual a lo que se obtiene cuando se usa *{author}* y *{maketitle}*, sin usar esos comandos; esto desemboca en problemas de indefinición cuando requerimos extraer de forma automática el título de varios documentos.

La reducción de los grados de libertad en la estructura que debe tener un tipo determinado de documentos, hace plausible extraer información de varios documentos automáticamente, por ejemplo para crear bases de datos o listados con información sobre una biblioteca de artículos. Los documentos XML son archivos de texto, que en principio están en código *Unicode*, pero se pueden usar otros alfabetos estandarizados como el *latin-1*. XML establece un conjunto de cinco caracteres especiales:

```
{ < ,  > , " , ' ,  &}
```

su utilización de manera textual, para evitar que sean interpretados de forma especial, require representarlos respectivamente con el siguiente conjunto de secuencias,:

```
{&lt; ,  &gt; ,  &quot; ,  &apos;  &amp; }
```

Cada marca tiene un nombre, por ejemplo *<figura>*, y puede tener asociados uno o más atributos:

```
<figura archivo="foto1.jpg" tipo="jpeg">
```

Una diferencia importante con SGML, y en particular con HTML, radica en que los nombres de las marcas y de sus atributos las mayúsculas y minúsculas son distinguibles; *<a>* y *<A>*

son dos marcas diferentes, es habitual usar únicamente minúsculas para los nombres de las marcas y de sus atributos. Otra diferencia sobresaliente respecto a SGML es que en XML ninguna marca se puede dejar abierta; o sea, por cada marca de apertura, por ejemplo * *, deberá existir su marca de cierre correspondiente, * *, que delimita el contenido de la marca. En el siguiente ejemplo: *<titulo>La mente Nueva del Emperador</titulo>*, el contenido de la marca *titulo* esta claramente delimitado entre *<titulo>* y *</titulo>*. Si una marca cualquiera no contiene ningún texto, por ejemplo *<hr></hr>*, se puede abreviar de la siguiente forma: *<hr/>*, la primera notación también es válida, en cambio escribir únicamente *<hr>* o *<hr/>* equivale a un error. La estructura de un documento XML es definido y validado por un esquema de tipificado, un documento XML es por completo transformable en otro documento XML. Su sintáxis permite reconocer y particionar su estructura por la presencia de los paréntisis angulares (*<>*).

En la expresión XML:

```
<persona>Alan Turing</persona>
```

el contenido de la marca *persona* esta claramente delimitado entre *<persona>* y *</persona>*.

Por lo lo tanto XML es textual, estructurado, y extensible. Comparado con lenguajes de alto nivel, permite representar una gran variedad de contenidos, XML tiene una sintaxis genérica que permite extenderlo, ya que su lenguaje (vocabulario y gramática) se puede redefinir, y al igual que en una expresión regular, permite incluir opciones de composición. Los principales componentes de un documento XML son los nodos mismos que, dado su caracter de contención gerárquica, se pueden representar mediante un árbol. Este modelo es objeto de una definición muy precisa, denominado tipo o esquema, con la finalidad de permitir a los lenguajes de programación manipular y procesar documentos XML. Los tipos fundamentales de nodo, que podemos identificar se presentan en el ejemplo siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
  <!-- '''Comentario''' -->
  <elemento-documento xmlns="http://ejemplo.org/" xml:lang=";sp">
    <elemento>Texto</elemento>
    <elemento>elemento repetido</elemento>
    <elemento>
      <elemento>Jerarquia recursiva</elemento>
    </elemento>
    <elemento>Texto con <elemento>elemento</elemento>mezclado</elemento>
    <elemento/><!-- elemento vacio -->
    <elemento atributo="valor"></elemento>
  </elemento-documento>
```

El procesamiento primario de un documento XML incluye su validación, contra un sistema de tipificado (DTD, XML Schema ...), así como la consulta de la información que contiene.Con el objetivo de evaluar la eficiencia de los algoritmos que realizan estas funciones, es útil conocer

las cotas teóricas de complejidad, donde nos ubica su desempeño. Este estudio se propone examinar y evaluar tales cotas, en particular ponemos a prueba el sistema DTD para la tipificación de documentos, y a XPath para evaluar las cotas del proceso de consulta.

2.1.1. Definición del Tipificado de Documento DTD

Las posibles marcas que pueden aparecer en un documento XML y los atributos que estas pueden tener, son definidos en un apartado al inicio del mismo documento, o bien en un archivo por separado, en ambos casos llamado *Definición del Tipo de Documento* o DTD (Document Type Definition). Todo documento XML debe indicar al comienzo el DTD que se aplica por medio de una marca *<!DOCTYPE>*; por ejemplo:

```
<!DOCTYPE xbel PUBLIC "+//IDN python.org//
DTD XML Bookmark Exchange Language 1.0//
EN//XML" "http://www.python.org/topics/xml/dtds/xbel-1.0.dtd">
```

Esta leyenda indica que lo que viene a continuación en el documento es una marca *xbel* (con todas sus posibles submarcas), que ha sido definida en un DTD que se llama *XML Bookmark Exchange Language 1.0*. La palabra clave PUBLIC precede al nombre oficial que se le ha dado al DTD respectivo; en este caso esa indicación nos da alguna información adicional al nombre del DTD: el símbolo + indica que es un DTD reconocido por alguna entidad oficial, en este caso python.org como lo indica la palabra clave IDN, el lenguaje usado en el DTD es el inglés (EN) y la sintaxis usada es sintaxis XML. Realmente el nombre que viene entre comillas después de PUBLIC es algo arbitrario, pero como en cada sistema existe un catálogo SGML que identifica los DTD disponibles en el sistema, lo importante es usar exactamente el nombre que aparezca en el catálogo. Y para aquellos documentos que usen el mismo DTD puedan ser transportables entre sistemas conviene usar la identificación exacta sugerida por el autor del DTD.

Después del identificador público (lo que está entre comillas después de PUBLIC) puede venir un identificador del sistema indicando el camino y nombre del archivo donde se encuentra el DTD; en el ejemplo anterior el identifcador del sistema es un URL que indica donde se puede encontrar el DTD usado. Un archivo DTD define siempre una o más estructuras jerárquicas, con una marca principal, o padre, compuesta por otras marcas, o hijos. Un documento DTD define siempre una o más estructuras jerárquicas, con una marca principal, o padre, compuesta por otras marcas, o hijos. La estructura de un DTD simple, con un elemento principal <article>. Dentro del elemento principal pueden aparecer otros elementos: <artheader>, <sect1>y <index>, y estos a su vez se componen de otros elementos.

La estructura de un DTD simple, con un elemento principal <books>. Dentro de ese elemento principal pueden aparecer otros elementos: <book>, y estos a su vez se componen de otros elementos <title>, <autor>, <year>, etc.,un documento XML construido a partir de esta estructura sería el siguiente:

```
<books>
```

```
    <book>
        <title>Elements of Finite Model Theory</title>
        <author>Leonid Libkin</author>
        <year>2004</year>
        <publisher>Springer</publisher>
        <rank>5/5</rank>
    </book>
    <book>
        <title>Animal Farm</title>
        <author>George Orwell</author>
        <year>2003</year>
        <publisher>Penguin Books</publisher>
    </book>
</books>
```

El DTD puede diseñarse de forma tal que haga obligatorio el uso de algunos sub-elementos y limitar el número de veces que un elemento puede aparecer y el orden de los elementos. De esta forma el DTD puede ser bastante flexible o tan exigente como se requiera, para forzar a los autores a ceñirse a una determinada gramática. Un documento XML que especifique el DTD usado y siga las reglas en él definidas, se dice que es un documento XML válido. Se pueden también crear documentos que no especifiquen ningún DTD pero que sigan las reglas mínimas del XML; en este caso el documento XML se denomina *conforme* (conforming); existen algoritmos para comprobar si un documento es válido, comprobar que el DTD existe y que la estructura del documento respeta las reglas definidas por el DTD.

2.1.2. Entidades

En la sección anterior ya hablamos de un tipo de entidades que se usan para representar caracteres adicionales al alfabeto usado. La únicas entidades de ese tipo que están predefnidas en XML son las cinco que ya mencionamos: <, >, ",&. Cualquier otra entidad adicional que queramos usar tendrá que estar definida previamente en el DTD usado. Por ejemplo, si estamos usando el alfabeto latin-1, que incluye el caracter ©, pero no sabemos como obtenerlo con el teclado, podremos definir una entidad ©. La definición se hace usando la marca ENTITY, de la siguiente forma

```
< ! ENTITY copy "&#169;" >
```

El número decimal 169 es el código que le corresponde al caracter ©en el alfabeto latin-1; también podríamos haber usado la representación hexadecimal #xA9;. La defnición de una entidad como la anterior puede ya formar parte del DTD, o puede ser adicionada por el autor del documento XML, dentro de la propia declaración del DTD del documento. Por ejemplo la fuente de un manual puede establecerse en un archivo XML con la siguiente definición de tipo de documento:

```
<!DOCTYPE article PUBLIC "-//laespiral.org//DTD LE-document 1.1//EN"
"LE-document-1.1.dtd" [<!ENTITYcopy "&#169;">] >
```

Entre los paréntesis cuadrados pueden ir varias definiciones de entidades. El valor de una entidad no está limitado a ser un caracter, sino que puede ser cualquier texto. Por ejemplo si definimos la entidad &qed; de la siguiente manera

```
<!ENTITY qed "Que es lo que queríamos demostrar">
```

Cada vez que en el documento se escriba &qed;, será substituido por el texto "Que es lo que queríamos demostrar". Una entidad se puede usar también para insertar el contenido completo de un archivo en un punto del documento, si se define de la siguiente forma

```
<!ENTITY nombre SYSTEM "archivo.txt">
```

El archivo puede contener inclusive marcas y cualquier otro texto que sea válido en el punto donde aparezca &nombre;. Existen otro tipo de entidades internas, que no pueden ser usadas en un documento XML sino únicamente dentro de un DTD, estas comienzan por el caracter especial % en vez de &.

2.1.3. Archivos XML

A los archivos XML se les suele dar un nombre terminado en .xml para identificarlos. Esto es simplemente una convención para los usuarios; el estándar XML 1.0 indica que para identificar un archivo como XML es necesario que la primera línea tenga el siguiente contenido

```
< ?xml version="1.0" ? >
```

Dentro de esta marca puede ir otra información adicional. El alfabeto usado por defecto en los archivos XML es el Unicode; para documentos en español será mas conveniente usar el alfabeto latin-1, lo cual se logra usando el atributo encoding de la marca xml

```
<? xml version="1.0" encoding="iso-8859-1" ?>
```

2.2. Construcción de un DTD

El DTD suele estar dentro de un archivo con extensión dtd, pero puede incluso ser definido dentro de la propia marca DOCTYPE en el documento XML. Veamos un ejemplo muy simple de un archivo XML que incluye también el DTD

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE cd[
<!ELEMENT cd (titulo, artista, pista+)>
<!ATTLIST cd fecha CDATA #IMPLIED>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT artista (#PCDATA)>
<!ELEMENT pista (#PCDATA)>
]>
<cd fecha="2001">
<titulo>Odisea 2001</titulo>
<artista>Opera Real de Londres</artista>
<pista>Apertura</pista>
<pista>Opus 12 (versión instrumental).</pista>
<pista>Allegreto(versión extendida).</pista>
</cd>
```

El elemento principal definido en el DTD es cd, el cual tiene que tener inicialmente una marca titulo, seguida de una marca artista y finalmente seguida de una o mas marcas pista; el símbolo más al lado de la marca pista, en la defnición del elemento *cd*, indica que tiene que aparecer por lo menos una vez. Otros modificadores usados son *, que significa cualquier número de veces incluyendo cero, e ? que indica que puede no aparecer o aparecer a lo sumo una vez; sino aparece ningún modifcador, la marca respectiva debe aparecer exactamente una vez. Si queremos que el orden de los subelementos título y artista de cd pueda ser arbitrario podemos usar la siguiente construcción:

```
<! ELEMENT cd ((titulo | artista)*, pista+) >
```

Pero en este caso estaríamos permitiendo que aparezca más de un título o autor (o ninguno). El elemento cd acepta un atributo llamado fecha. Para indicar que el contenido de un atributo o elemento puede ser una combinación de caracteres del alfabeto usado, empleamos la palabra clave CDATA, en el caso de los atributos, y #PCDATA en el caso de los elementos; otras posibilidades para el tipo de datos de los atributos son NMTOKEN, cuando solo puedan tener valores numéricos, ID cuando sea un código de identificación que tenga un valor único, e IDREF cuando tenga que ser una referencia a un código de identificación ya existente. Los elementos también pueden incluir la palabra clave EMPTY cuando se trate de elementos que no pueden tener ningún contenido. La palabra clave #IMPLIED indica que el atributo es opcional; si fuera obligatorio se usaría en su lugar #REQUIRED, y si quisiéramos especificar una lista de posibles valores, se pondrían entre paréntesis, separados por barras verticales, y después de los paréntesis se escribiría el valor por defecto. Como normalmente estaremos interesados en crear varios documentos con estructura semejante, es mejor colocar el DTD en un archivo separado. El DTD del ejemplo anterior, dentro de un archivo aparte, quedaría así:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

```
<!ELEMENT cd (titulo, artista, pista+)>
<!ATTLIST cd fecha CDATA #IMPLIED>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT artista (#PCDATA)>
<!ELEMENT pista (#PCDATA)>
```

Dentro del fichero del DTD se pueden usar entidades para simplificar su escritura. Por ejemplo, una sección del DTD LE-document-1.1.dtd es la siguiente

```
<!ENTITY % listtype " itemizedlist | orderedlist | variablelist |
simplelist | programlisting | figure | form | table " >
<!ELEMENT article (artheader, (para | sect1 | %listtype;)*, bibliography?) >
<!ATTLIST article lang CDATA #IMPLIED
xreflabel CDATA #IMPLIED
id ID #IMPLIED
parentbook IDREF #IMPLIED>
```

El DTD completo se puede ver en:

`http://www.laespiral.org/xml/styles/LE-document-1.1.dtd`

La creación de un DTD es una tarea que requiere cuidado, no por la parte del código que se tiene que escribir, sino porque el diseño de la estructura jerárquica y las marcas usadas es crucial en el éxito de un DTD. En algunas aplicaciones se necesita mucha experiencia para tomar las decisiones acertadas sobre el diseño del DTD. Afortunadamente existen muchos grupos de expertos trabajando en la creación de DTDs públicos que pueden ser usados libremente.

Un líder en el campo de creación de DTDs es el propio consorcio W3C, que ya tiene varios DTDs disponibles. Algunos de ellos son: SVG para gráficos vectoriales, MathML para escritura matemática y XHTML que es una versión XML del DTD de HTML. Otros DTDs importantes creados por otros grupos son DocBook (originalmente en SGML, pero ya en versión XML) para escribir libros, especialmente manuales de software, BioML y BSML para biología, CML para química, AML y AIML para astronomía y TMX para traducciones.

Lenguajes de Páginas de Estilo

Un elemento importante para poder estructurar la información de un documento es separar el contenido del documento de su formato. Quien esta familiarizado con LaTeX sabe que una de sus principales ventajas es que permite a los autores concentrarse en el contenido del documento, sin tener que preocuparse mucho con la forma como será presentado. El formato que se usa para presentar el contenido está definido en otro archivo, que define la *documentclass*, que ha sido preparado por un experto, de manera que cualquier autor puede producir documentos de elevada calidad tipográfica sin mucho esfuerzo.

En HTML y XML ha habido también un esfuerzo por separar el contenido del formato y dar la posibilidad de reutilizar un formato pre-definido. El formato usado lo define otro archivo llamado una hoja de estilo (Style Sheet) usando un lenguaje propio para páginas de estilo.

Cascading Style Sheet Language (CSSL).

CSSL es el lenguaje para páginas de estilo desarrollado por el W3C para ser usado en páginas HTML. Un documento XML puede también hacer uso de una página de estilo CSS, en forma semejante a como se hace en una página HTML. Pero tenemos otro estándar mas reciente para páginas de estilo (XSL).

eXtensible Style-sheet Language (XSL)

El lenguaje de páginas de estilo que ha sido desarrollado por el consorcio W3C, para dar formato a los documentos XML, se llama XSL. Una página de estilo XSL permite modificar un documento XML, produciendo varios un resultado que puede estar en varios formatos diferentes incluyendo el propio XML y HTML.

Una página de estilo XSL es también un documento XML que usa el DTD xsl:stylesheet. El comienzo de una página usada para producir HTML podría tener el siguiente contenido

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE xsl:stylesheet[<!ENTITY nbsp " ">]>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" encoding="iso-8859-1"
doctype-public="//w3c//dtd html 4.0 transitional//en"/>
```

En este caso se ha defnido una entidad adicional que será usada en el HTML producido, y se ha especificado la información que deberá aparecer en la marca DOCTYPE del archivo HTML que se genere. El espacio de nombres usado; o sea la especificación de marcas usadas en XSL, se ha definido con el atributo xmlns:xsl.

Todas las marcas de XSL comienzan con la secuencia xsl:. La marca básica que realiza el procesamiento del archivo XML, es la marca *<xsl:template>* que define la plantilla que se debe usar para producir la salida de datos. Veamos un ejemplo

```
<xsl:template match="itemizedlist">
<ul>
<xsl:apply-templates/>
</ul>
</xsl:template>
```

Esta plantilla será aplicada cada vez que aparezca una marca *<itemizedlist>* en el documento XML; al texto que aparezca entre *<itemizedlist>* y la correspondiente *</itemizelist>*. En este caso se usará la marca que crea listas de items en HTML: *<ul>*. La marca *<xsl:apply-templates/>* hace que el procesamiento continúe, aplicando todas las otras plantillas que sean relevantes al texto que se ha seleccionado (el contenido de <itemizelist>). Algo importante que

se debe tener en cuenta es que aunque queramos que en el archivo HTML de salida aparezca una marca vacía como por ejemplo *
*, en la plantilla se debe escribir *
*, pues la plantilla hace parte de un documento XML; en la salida aparecerá *
* pues el resultado se presenta en HTML.

Si la página de estilo tuviera que producir LaTeX en vez de HTML, la plantilla correspondiente a la anterior sería

```
<xsl:template match="itemizedlist">
\begin{itemize}
<xsl:apply-templates/>
\end{itemize}
</xsl:template>
```

2.3. Herramientas para XML

Existen varias herramientas disponibles en software libre para trabajar con archivos XML. Muchos programas usan la plataforma XML como un medio de intercambiar información o como especificación para escribir los archivos de configuración.

Catálogo SGML

En un sistema en el que se usa XML para crear documentos, conviene que exista un catálogo de los DTD disponibles y la información de donde encontrarlos en el sistema. En Debian GNU/Linux, las herramientas para crear y mantener el catálogo vienen dentro del paquete sgml-base; este paquete incluye el programa install-sgmlcatalog que será usado por otros paquetes que instalen DTDs, para actualizar el catálogo, que se encuentra localizado en/etc/sgml.catalog.

Edición de Archivos XML

Existe un paquete psgml que define un modo XML para el editor Emacs. Emacs junto con psgml es bastante útil para editar archivos XML. Para comenzar a escribir un documento, el primer paso es crear un archivo con extensión xml y con el siguiente contenido en las dos primeras líneas:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE article PUBLIC "-//laespiral.org//DTD LE-document 1.1//EN"
"http://www.laespiral.org/xml/styles/LE-document-1.1.dtd">
```

En este caso vamos a usar el DTD *LE-document*, y vamos a utilizar caracteres *latin-1*. El elemento principal en el documento será <article>.

La extensión xml del nombre del archivo ha hecho que emacs entre en el modo XML depsgml, como se puede ver en el centro de la línea de estado (la línea negra en la parte inferior) y por la aparición de varios menús adicionales para trabajar con XML. El modo XML se ha encargado también de leer la línea que define el DTD y ha cargado el archivo LE-document-1.1.dtd (si existe una copia local del DTD, se puede substituir la URL por el camino completo de esa copia).

En el despliegue del documento se puede ver que el DTD ya ha sido leído y analizado, pues ya ha sido identificado el elemento principal del documento: "article"; también puede ver alguna información sobre el DTD y los elementos que define, en el menú DTD que presenta psgml en Emacs. Si no aparece esa información, por ejemplo si comenzó a escribir las dos primeras líneas en un archivo vacío, tendrá primero que asegurarse de que está usando el modo XML, y después seleccionar la opción "Parse DTD" en el menú DTD (o si prefiere puede usar la secuencia "C-cC-p").

Después de estar en modo XML y de haber seleccionado un DTD, se puede usar una opción de menú muy útil que nos permite escribir el documento rápidamente; se trata de la opción Insert Element, en el menú "Markup". Esta opción nos muestra una lista de los elementos que son permitidos en el punto donde se encuentra el cursor; escogiendo un elemento en la lista, son introducidas las etiquetas exigidas por ese elemento.

Analizadores sintácticos de XML/XSL

Existen varios analizadores sintácticos de XML que permiten determinar si un documento XML es válido. Algunos ejemplos son nsgmls y rx. También existen programas que aplican páginas de estilo XSL para transformar documentos XML en otros formatos como por ejemplo HTML; tres ejemplos son Sablotron, Xalan y Libxslt.

Cocoon

El proyecto Apache tiene un grupo dedicado exclusivamente al desarrollo de herramientas XML. Ya han desarrollado programas Java para analizar y transformar XML (Xercesy Xalan) y un servelet llamado Cocoon, que procesa documentos XML y les aplica las transformaciones indicadas por una hoja de estilo XSL para producir HTML. De esta forma se puede configurar el servidor http de Apache para que genere código HTML dinámicamente, a partir de archivos fuente XML.

Document Object Model (DOM) y Standard API for XML (SAX).

Como ya mencionamos estos métodos se aplican para analizar sintácticamente un documento XML. El primer método, DOM, lee el documento completo e identifica su estructura jerárquica. El segundo método, SAX, va identificando las marcas a medida que va leyendo el documento. El segundo método es obviamente más rápido y consume menos recursos, pero tiene la desventaja de que cada vez que aparece una marca debe decidir que hacer con ella, y no puede regresar para atrás en el documento.

SAX se desarrolló con aplicaciones de servidor en mente; el servidor debe suministrar rápidamente el resultado de transformar un documento XML. DOM fue desarrollado con aplicaciones de cliente en mente; por ejemplo un editor de XML necesita poder navegar en cualquier dirección la estructura del documento; en este caso el método SAX no es muy útil. Existen varias bibliotecas disponibles que implementan un u otro método en varios lenguajes de programación diferentes. A continuación se presenta un ejemplo de un programa Perl que usa el módulo XML: DOM para sacar información de un archivo XML:

```
#! /usr/bin/perl
use XML::DOM;
my $archivo = 'archivo.xml';
my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ($archivo);
my $titulo = &extraer($doc->getElementsByTagName ("titulo"));
my $autor = &extraer($doc->getElementsByTagName ("autor"));

sub extraer

{
my (@elementos) = @_;
my $elemento = $elementos[0]->toString;
$elemento =- s/-[->]*>\nf//;
$elemento =- s/\nf\s*<[-<]*$//;
return $elemento;
}
```

Este programa lee el archivo ".xml" y extrae la información de las marcas titulo y autor. La subrutina extrae la información y elimina las marcas que delimitan cada elemento.

2.4. Aplicaciones de XML

Son muchas las aplicaciones de XML; y con la existencia de bibliotecas para producir, analizar y transformar XML, disponibles para varios lenguajes de programación, cada día se usa más el XML en varios campos muy diversos.

Preparación de Documentos

XML es un sistema muy útil para preparar documentos, un manual puede escribirse en XML, al cual se le puede aplicar una hoja de estilo XSL para generar un archivo LaTeX. Existen varias ventajas de usar XML en vez de producir directamente un archivo LaTeX.

El DTD orienta al autor en los pasos que debe seguir, al usar un editor para archivos XML e introducir el elemento inicial, nos aparece la estructura mínima que debe tener el documento; y

en cada parte del documento podemos consultar una lista de las posibles marcas que se pueden usar en esa sección. XML permite ser más exigente respecto a la estructura del documento, lo que permite una mayor uniformidad entre diferentes documentos.

Existen programas que permiten revisar un documento XML rápidamente y descubrir errores de sintaxis o de la estructura del documento. en LaTeX se puede revisar la sintaxis, pero descubrir fallas en la estructura es más difícil. Si por ejemplo el autor definió una sub-sección antes de haber definido alguna sección, o si repitió el título en el medio del documento, el resultado continúa siendo un archivo LaTeX válido y ese tipo de errores son difíciles de descubrir de forma automática.

XML es más fácil de transformar en otros formatos. Por ejemplo, pasar de XML a HTML puede ser realizado fácilmente con una página de estilo XSL. Sin embargo cuando se trata de documentos con bastante contenido matemático, aún no existe ningún DTD que permita escribir ecuaciones con la facilidad y el poder disponibles en LaTeX y TeX.

Creación de Páginas Web

Escribir una página HTML es una tarea fácil. Pero cuando queremos construir un site completo, la labor es mucho mas ardua porque es necesario tener un buen sistema de navegación entre las páginas e intentar crear una imagen de marca que sea consistente en todas las páginas. A medida que el site crece, se va volviendo mas complicado su mantenimiento, y un pequeño cambio puede implicar tener que actualizar varias páginas.

Algunas soluciones adoptadas frecuentemente son PHP (Hypertext Preprocessor) o SSI (Server Side Includes) para introducir información de forma dinámica en las páginas. De esta forma si hay información que es actualizada frecuentemente, esta puede entrar en forma automática en las páginas.

Sin embargo, no es muy conveniente convertir todas las páginas en dinámicas, ya que el site puede llegar a tornarse muy lento debido a una sobrecarga del servidor. Las páginas dinámicas también pueden ser difíciles de modificar porque pueden ser auténticos programas que quien no esté muy familiarizado con su funcionamiento no los podrá modificar fácilmente.

XML puede ayudar a resolver estos problemas, separando el contenido de la presentación, se puede mantener la información mínima y necesaria en las páginas, convirtiéndolas en mas fáciles de modificar y actualizar. La presentación y los menús o las barras de navegación pueden estar contenidos en una página de estilo general. Con un simple comando make se pueden generar páginas estáticas HTML a partir de las fuentes XML, cada vez que existan cambios.

La dificultad de este método está en la construcción de un DTD adecuado para páginas Web, por suerte el trabajo ya está hecho: se trata de XHTML, que es un DTD XML que describe básicamente todas las funcionalidades del lenguaje HTML 4.0, pero con sintaxis XML. Quien esta acostumbrado a trabajar con HTML, solo tiene que tomar en cuenta unas pocas reglas para hacer la transferencia a XHTML:

- Nunca usar mayúsculas en los nombres de las marcas

- No dejar ninguna marca abierta; es necesario cerrarlas con la marca respectiva </.>, o si se trata de una marca vacía, se puede usar por ejemplo
- Usar siempre comillas para delimitar el valor de los atributos

Organización de Información con Resource Description Framework (RDF)

RDF es un DTD orientado a la descripción de recursos. Con el rápido crecimiento de la WWW, la cantidad de información disponible en campos muy variados es bastante amplia. Un problema complicado es como clasificar la información disponible para poder encontrarla cuando es necesaria. Han existido intentos de crear meta-catálogos de documentos disponibles en la Web, pero ha resultado ser una tarea muy complicada ya que las páginas Web aparecen y desaparecen con mucha facilidad. Otro enfoque han sido los motores de búsqueda que recorren la Web clasificando información de forma automática. La dificultad existente es que sin un buen resumen de lo que contiene un documento, una búsqueda automática puede no ser muy útil. Ha habido intentos de facilitar la tarea de clasificación de los robots, usando por ejemplo las marcas META en el encabezado de las páginas Web.

El RDF ha sido concebido con este tipo de problemas en mente. Su objetivo es la descripción de recursos disponibles; y normalmente se destina al intercambio de información entre sistemas, más que a suministrar contenido. Un ejemplo de aplicación es en los servidores de noticias, como por ejemplo Slashdot y Barrapunto. El listado siguiente muestra lo que se obtiene si seleccionamos una sección de Barrapunto, en este caso la sección “La Espiral” y accedemos a la URL

```
http://barrapunto.com/laespiral.rdf
```

(solamente se muestra la parte inicial del listado, para dar una idea de como es)

```
<?xml version="1.0"?>

<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns="http://my.netscape.com/rdf/simple/0.9/">

 <channel>
 <title>Barrapunto</title>
 <link>http://barrapunto.com/</link>
 <description>La informaci&#243;n que te interesa</description>
 </channel>

 <image>
 <title>Barrapunto</title>
 <url>http://images.barrapunto.com/topics/topicbarrapunto.png</url>
```

```
<link>http://barrapunto.com/</link>
</image>

<item>
<title>Se reactiva La Espiral</title>
<link>http://barrapunto.com/article.pl?sid=03/09/30/0058220</link>
</item>

<item>
<title>Traducci&#243;n de descripciones de paquetes en Debian:
                    unificaci&#243;n</title>
<link>http://barrapunto.com/article.pl?sid=01/09/14/0244205</link>
</item>
```

Como se puede ver, el resultado es un documento XML que usa el DTD RDF; realmente es una pequeña implementación de RDF, llamada RSS (RDF Site Summary), pues RDF puede ser mucho mas complejo. Este archivo RDF será muy fácil de manipular con los procesadores de XML/XSL y podrá ser usado para extraer por ejemplo los titulares de las noticias en la sección.

Existen varios programas que ayudan a crear o procesar RSS. En Debian el paquete libxml-rss-perl trae un módulo Perl que se puede usar para ese propósito.

Archivos de Configuración

XML es una muy buena opción para escribir archivos de configuración de programas. La existencia de bibliotecas optimizadas para extraer información y modificar documentos XML facilita la tarea del programador.

Un ejemplo típico son los archivos de confguración usados en Glade. Glade es un programa para construir interfaces gráficas de usuario (GUI) que usan las bibliotecas gráficas GTK+. Glade tiene una interfaz gráfica fácil de usar, donde se pueden definir las ventanas que usará el programa por construir y se le pueden ir colocando diferentes “widgets”. El resultado después de definir la GUI del programa se resume en un archivo XML con el nombre del programa y con extensión “.glade”. Si más tarde se quiere modificar la interfaz gráfica del programa, glade leerá ese archivo y analizándolo recuperará toda la información que necesita para volver a representar la interfaz gráfica del programa.

Glade trae un ejemplo que consiste en un editor de texto. Las primeras líneas del archivo editor.glade son así:

```
<?xml version="1.0"?>
<GTK-Interface>
<project>
<name>Glade Text Editor</name>
```

```
<program_name>glade-editor</program_name>
<directory></directory>
<source_directory>src</source_directory>
<pixmaps_directory>pixmaps</pixmaps_directory>
<language>C</language>
<gnome_support>False</gnome_support>
<gettext_support>True</gettext_support>
```

Notemos que no usa ningún DTD, cuando el archivo XML se crea y modifica por un programa, normalmente no es necesario validarlo contra un DTD, pues si el programa ya ha sido depurado, los archivos producidos tendrán siempre la estructura esperada.

Bases de Datos

Los documentos XML son una buena interfaz para proporcionar información a una base de datos, o para almacenar copias de partes del contenido de la base de datos, en archivos de texto. Cada campo en una tabla de la base de datos se puede hacer corresponder al contenido de alguna marca XML. Por ejemplo podemos tener una subrutina en lenguaje *Perl*, con el que se utiliza DOM para extraer información de un archivo XML y actualizar una base de datos SQL.

Procesamiento Distribuido usando Simple Object Access Protocol (SOAP)

SOAP es un protocolo usado para ejecutar comandos en servidores remotos. La información enviada al servidor remoto y el resultado de la ejecución del comando se envían en documentos XML.

Los mensajes SOAP con base en XML tienen el potencial de transformar la forma en que programamos aplicaciones distribuidas, y como intercambiamos datos. SOAP es un mecanismo que permite llamadas remotas a procedimientos, o la invocación de métodos remotos, con la misma funcionalidad que otras opciones ya establecidas y maduras como DCOM (Distributed COM), o IIOP (Internet Inter-ORB Protocol). De hecho SOAP es una plataforma neutral, un lenguaje neutral y sin dependencia en algún modelo de objeto en particular, por consiguiente, una aplicación distribuida que incluya SOAP puede compartirse entre múltiples sistemas operativos, puede estar construida a partir de objetos de diferentes fabricantes, escritos en diferentes lenguajes, y basado en modelos de objetos diferentes.

SOAP sobre http, por ejemplo, permite aplicaciones distribuidas a partir de dos tipos de escenarios de comunicación en la Web: consulta/respuesta (request/response) y envía-y-olvida (fire-and-forget): En el primero el emisor invoca un método de llamada en un objeto remoto, sin requerir un valor de contestación, en el segundo un objeto puede establecer una comunicación bidireccional con el emisor invocando un método de llamada y recibiendo un valor de contestación.

3. COMPLEJIDAD, LÓGICA, Y AUTÓMATAS

El presente capítulo puntualiza el marco teórico que requiere el estudio de las cotas de la complejidad que caracterízan los algoritmos básicos de verificación y de consulta de documentos XML recomendados por el 3WC. Los conceptos por revisar se ubican en tres áreas de la Ciencia de la Computación: Teoría de la Complejidad, Lógica, y Autómatas.

3.1. Complejidad Computacional

Cuando tenemos que resolver computacionalmente un problema, es posible encontrar varios algoritmos eficaces y adecuados; evidentemente deseamos seleccionar el mejor. El enfoque *a priori*, consiste en determinar la cantidad de recursos necesarios que su ejecución requiere, en términos de espacio de almacenamiento y del tiempo de proceso, que cada uno de los algoritmos necesita como función del tamaño de la estructura de entrada por computar. El tamaño de tal estructura corresponde formalmente al número de *bits* que se necesitan para representar a la estructura en una computadora, utilizando algún esquema de codificación definido y razonablemente compacto, consideraremos por tamaño a cualquier entero n que mida de alguna manera el número de componentes de la estructura de entrada, por ejemplo; si estamos refiriéndonos a ordenaciones, mediremos el tamaño por el número de componentes que hay que ordenar; si se trata de un grafo, medimos su tamaño por el número de nodos y aristas; en el caso de un arreglo, el "tamaño del problema" corresponde al número de elementos que contiene. En el caso de problemas con números en lugar del tamaño hablaremos del *valor* del ejemplar. Sí por otra parte, tal como sucede en nuestro análisis, deseamos medir la eficiencia de un algoritmo, en términos de los recursos que necesita para llegar a una respuesta, entonces está claro que no podemos utilizar una unidad de tiempo determinada, puesto que no se dispone de una computadora estándar a la cual se pudiesen referir todas las medidas; una respuesta a este problema es el *principio de invariancia*, que afirma que dos implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia más que en una constante multiplicativa. Si esta constante fuese, por ejemplo nueve, entonces sabemos que si la primera implementación requiere 1 milisegundo para resolver estructuras de cierto tamaño, entonces la segunda implementación (en una máquina diferente o en un lenguaje distinto) no requerirá más de 9 milisegundos para resolver los mismos casos.

En general, si dos implementaciones del mismo algoritmo necesitan $t_1(n)$ y $t_2(n)$ segundos , respectivamente, para resolver un caso de tamaño n, entonces siempre existen constantes positivas c y d tales que $t_1(n) \leq ct_2(n)$ y $t_2(n) \leq dt_1(n)$ siempre que n sea suficientemente grande. En otras palabras, el tiempo de ejecución de cualquiera de las implementaciones está

acotado por un múltiplo constante del tiempo de ejecución de la otra. Este principio no es algo que se pueda demostrar, simplemente establece un hecho que puede ser comprobado por observación.

Los algoritmos se pueden categorizar con base en la cantidad de tiempo y de espacio en memoria que necesita su ejecución, es decir que tan rápido crece el tiempo y la memoria requeridos cuando el tamaño del problema crece asintóticamente. Si el tamaño del problema lo representamos por la variable n que puede tomar valores enteros no negativos, el tiempo de ejecución lo expresamos como la función $t(n)$, mientras que la memoria ocupada con la función $s(n)$, la eficiencia del algoritmo se evalúa a partir de la rapidez con que crecen estas funciones cuando n va tomando valores cada vez más grandes. La notación $\mathcal{O}$ y sus variantes, Ω, Θ y $\mathbf{o}$; nos permiten expresar convenientemente estas dimensiones temporal y espacial relativas de la complejidad del algoritmo en condiciones de crecimiento asintótico del tamaño del problema, a continuación las definimos:

Definición 3.1. *Sea* $\mathbf{N}$ *el conjunto de números enteros no negativos,* $n \in \mathbf{N}$*,* $f : \mathbf{N} \to \mathbb{R}^+ \cup \{0\}$*, y* $f(n)$ $\max\{\lceil f(n) \rceil, 0\}$*. Categorizamos el comportamiento relativo de un conjunto de funciones respecto a la función* f *con las notaciones* $\mathcal{O}(f)$ *"del orden con cota superior marcada por* f*",* $\Omega(f)$ *"del orden con cota inferior marcada por* f*",* $\Theta(f)$ *"del orden exacto de* f*" y* $\mathbf{o}$(f) *"del orden inferior a* f*" de acuerdo a:*

$$\mathcal{O}(f(n)) = \{g : \mathbf{N} \to \mathbb{R}^+ \cup \{0\} \quad | \quad \exists n_0, c \in \mathbf{N}. \quad \forall n \geq n_0 \quad |g(n)| \leq c \cdot |f(n)|\}. \tag{3.1a}$$

$$\Omega(f(n)) = \{g : \mathbf{N} \to \mathbb{R}^+ \cup \{0\} \quad | \quad \exists n_0, c \in \mathbf{N}. \quad \forall n \geq n_0 \quad |g(n)| \geq c \cdot |f(n)|\}. \tag{3.1b}$$

$$\Theta(f(n)) = \{g : \mathbf{N} \to \mathbb{R}^+ \cup \{0\} \quad | \quad \exists n_0, c, d \in \mathbf{N}. \quad \forall n \geq n_0, c \cdot |f(n)| \leq |g(n)| \leq d \cdot |f(n)|\}. \tag{3.1c}$$

$$\boldsymbol{o}(f(n)) = \{g : \mathbf{N} \to \mathbb{R}^+ \cup \{0\} \quad | \quad \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\}. \tag{3.1d}$$

□

Con base a estas definiciones, se pueden establecer las siguientes relaciones:

$$\Theta(f(n)) = \quad \mathcal{O}(f(n)) \quad \cap \quad \Omega(f(n)) \tag{3.2a}$$

$$\mathcal{O}(f(n)) \quad \supset \quad \Theta(f(n)) \quad \cup \quad \mathbf{o}(f(n)) \tag{3.2b}$$

Para contextualizar la complejidad computacional analicemos el juego de las 20 preguntas para adivinar un número. Su contricante selecciona un entero positivo menor que un millón,

que Ud. tiene que encontrar. Se le permite un máximo de 20 preguntas con respuesta del tipo *si* o *no*, que deberán ser contestadas sin ambigüedad por su adversario. Por ejemplo se puede preguntar *¿El número es primo?*. Si se está familiarizado con este juego, utilizará una estrategia de divide y vencerás para resolver el acertijo, dividiendo por dos, con cada una de las preguntas, el número de candidatos a número correcto, por consiguiente, la primera pregunta sería *¿Está el número entre 1 y 500,000?*. Se puede mostrar que siempre se hallará la respuesta con al menos las 20 preguntas permitidas; ya que el tamaño n del problema corresponde a una lista con 1,000,000 elementos, y la cantidad de preguntas a $f(n) = \log_2 n$, es decir $f(10^6) = \lceil \log_2 10^6 \rceil = 20$. Encontrar el algoritmo para resolver el problema que consistente en establecer las preguntas *suficientes* para deducir el número, es un tema que le corresponde al arte de la Algorítmica; sin embargo el que las veinte preguntas sean o no estrictamente *necesarias*, concierne a la Teoría de la Complejidad. Se puede demostrar que nuestro algoritmo de complejidad $\mathcal{O}(\log n)$ utilizará casi las 20 preguntas en la mayoría de los casos por adivinar; sin embargo, no se puede concluir, basándose en la incapacidad de este algoritmo para resolver siempre el acertijo con menos de veinte preguntas, que no sea posible hacerlo mediante un algoritmo más inteligente.

Considéremos un algoritmo formado por tres pasos: inicialización, procesamiento y finalización, supongamos que su procesamiento requiere recursos, en función del tamaño n del problema, de acuerdo a $\mathcal{O}(n^2)$, $\mathcal{O}(n^3)$ y $\mathcal{O}(n \log n)$, respectivamente. Entonces, el algoritmo completo necesita recursos de acuerdo a $\mathcal{O}(n^2 + n^3 + n \log n)$, dichos recursos son del orden de los recursos requeridos por la parte que consuma más cantidad de ellos, independientemente del tamaño de la entrada, es decir:

$$\mathcal{O}(n^2 + n^3 + n \log n) = \mathcal{O}(\text{máx}(n^2 + n^3 + n \log n)) = \mathcal{O}(n^3)$$

Al estudiar la complejidad de un algoritmo es importante empezar a simplificar y a formalizar, tanto como sea posible, cada uno de los conceptos que intervienen; de tal manera que estos sean claros y concisos; y puedan constituirse en un medio de análisis incremental. Esto se logra planteando el análisis de la complejidad en terminos de Máquinas de Turing, de Lenguajes y de arquitecturas de circuitos lógicos.

Los problemas que podemos resolver mediante un algoritmo se pueden caracterizar en aquellos que requieren un "si" o un "no" como respuesta y aquellos que requieren realizar un cálculo y dar un valor como respuesta. A los primeros se les denomina de decisión y a los segundos de cálculo, estos últimos pueden replantearse también como problemas de decisión al proponer un valor en su resultado, y preguntar si éste es o no correcto. Podemos abordar cualquier problema de decisión, pero para su procesamiento por una máquina necesitamos codificar su entrada en forma de una cadena compuesta a partir de un alfabeto de símbolos. De esta manera la estructura de datos básica es una cadena (string), todas las demás estructuras de datos se codifican y se representan mediante cadenas.

Definición 3.2. *Un **alfabeto** Σ es un conjunto finito de símbolos elementales, si y sólo si al descomponer una secuencia finita construida a partir de Σ. cada uno de ellos es completamente identificable de manera única.*

*Una **cadena** w es una secuencia finita de símbolos de un alfabeto Σ. Una cadena puede carecer de símbolos, a esta cadena ϵ se le llama **cadena vacia**.*

*El operador denominado **estrella de Kleene** sobre un alfabeto Σ, denotado por Σ^* es el conjunto de todas las cadenas, incluyendo la vacia, formadas a partir del alfabeto Σ.*

*Un **lenguaje** L es un conjunto de cadenas w sobre un alfabeto Σ. Por consiguiente podemos especificar un lenguaje infinito siguiendo el siguiente esquema:*

$L = \{w \in \Sigma^* | w \text{ tiene la propiedad } P\}$

*La **concatenación de los lenguajes** L_1 y L_2 definidos sobre un alfabeto Σ esta dada por:*

$$L = L_1 \circ L_2 = L_1 L_2 = \{w \in \Sigma^* | w = x \circ y \quad \text{para cualquier } x \in L_1 \text{ y } y \in L_2\}. \tag{3.3}$$

La operación estrella de Kleene sobre un lenguaje L, denotado por L^* es el conjunto de todas las cadenas obtenidas al concatenar cero o más cadenas de L, por consiguiente:

$$L^* = \{w \in \Sigma^* | w = w_1 \circ \cdots \circ w_k \quad \text{para cualquier } k \geq 0 \text{ y cualquier } w_1, \ldots w_k \in L\}. \tag{3.4}$$

3.1.1. Teoría de la Complejidad

La complejidad computacional tiene una trayectoria paralela al análisis y diseño sistemático de los algoritmos, considera globalmente todos los algoritmos posibles para resolver un problema dado, incluyendo aquellos en los que ni siquiera se ha pensado todavía. Estudiando un algoritmo podemos demostrar que el problema objeto de nuestro estudio se puede resolver en un tiempo que está en $\mathcal{O}(f(n))$ para alguna función $f(n)$ que intentamos reducir lo más posible y encontrar su cota inferior. Al emplear la complejidad, por otra parte, intentamos hallar una función $g(n)$ tan grande como sea posible para la cual podamos demostrar que *cualquier* algoritmo capaz de resolver correctamente nuestro problema en todos sus casos debe necesariamente requerir un tiempo que este en $\Omega(g(n))$. A esta función $g(n)$ se le da el nombre de *cota inferior* de la complejidad del problema en el peor de los casos. Quedamos satisfechos cuando $f(n) \in \Theta(g(n))$ dado que hemos encontrado el algoritmo más eficiente posible. Desafortunadamente, en el estado actual de conocimiento, estos casos existosos no se encuentran con frecuencia. Sin embargo, y como contraparte, de vez en cuando podemos encontrar incluso el número exacto de veces que es preciso realizar una operación (por ejemplo una comparación), para resolver el problema.

El objeto de estudio de la Teoría de la Complejidad, desde su enfoque clásico denominado también estructural, consiste en la clasificación de los problemas según la cantidad de los recursos computacionales aplicados a su resolución. Considera tanto los problemas de decisión como los de cálculo: por ejemplo, decidir si existe un camino más corto con valor inferior a un determinado valor entre dos vértices de una gráfica con pesos, o bien calcular la longitud o peso de ese camino. Agrupa los problemas en clases según los recursos utilizados por los algoritmos que permiten encontrar su solución. Las medidas de la complejidad se obtienen estableciendo una cota para el tiempo que toma la ejecución del cálculo o bien el espacio en memoria que se utiliza.

Otro punto de vista más reciente, analiza a la complejidad desde el ángulo descriptivo; en contraste al enfoque clásico, los problemas de decisión se representan como conjuntos de cadenas que conforman un lenguaje reconocido por una máquina. Una generalización natural consiste en interesarse en conjuntos de estructuras finitas arbitrarias; por ejemplo las estructuras de gráficas, habitualmente codificadas en la máquina mediante cadenas. La complejidad descriptiva caracteriza los recursos necesarios para describir un problema dentro de un marco lógico. Las medidas de la complejidad no se asocian con los medios materiales para su ejecución, sino a lenguajes lógicos que permiten expresar una clase de problemas; una de las medidas que utiliza es la complejidad de los cuantificadores de la fórmula que define al problema. Lo que aporta este enfoque es la demostración que existe una correspondencia precisa con los resultados obtenidos por el enfoque clásico, poniendo en evidencia las caracterizaciones lógicas de las principales clases de complejidad en tiempo y en espacio.

Una tercera vía de análisis de la complejidad computacional es la desarrollada al clasificar los problemas, definidos como predicados o funciones booleanas, por las dimensiones de los circuitos lógicos que se necesitan ensamblar para resolverlas. Este esfuerzo se ha formalizado en un campo denominado *teoría de la complejidad de los circuitos*, la importancia de sus estudios ha cobrado, durante la última década, una relevancia creciente debido a la relación que existe entre el tamaño y profundidad de los circuitos lógicos, con el tiempo de proceso en una computadora con arquitectura de procesamiento en paralelo. Las clases de complejidad definidas bajo este enfoque encuentran relaciones y correspondencias con las establecidas por la teoría de la complejidad clásica.

La materia básica de la Teoría de la Complejidad consiste en determinar la cantidad de recursos de cómputo, desde el punto de vista clásico, requeridos para resolver un problema, así como clasificar estos problemas de acuerdo al grado de dificultad que presenta su solución. Los recursos considerados son el tiempo, la memoria (espacio) y la circuitería (o hardware). El reto principal de la teoría de la complejidad consiste en demostrar las cotas inferiores de complejidad, es decir, definir que problemas no se pueden resolver al menos que apliquemos grandes cantidades de recursos. A pesar que es relativamente fácil probar que existen problemas inherentemente difíciles, resulta que es mucho más difícil probar que la solución de problemas relevantes en la teoría computacional presenta un elevado grado de complejidad. La teoría de la complejidad ha tenido más éxito proporcionando evidencias duras de intratabilidad (intractability) de los problemas, con base en conjeturas convincentes y ampliamente aceptadas, los argumentos matemáticos de la intratabilidad dependen de las nociones de reducibilidad (reducibility) y de complitud (completeness), antes de revisar estos conceptos definiremos el concepto de clase de complejidad.

A pesar que pocas cotas inferiores generales se consideran como suficientemente interesantes para que sean traducidas en cotas inferiores concretas, la teoría de la complejidad ha contribuido (1) con una forma de dividir el mundo computacional en clases de complejidad, y (2) ha evidenciado que estas clases de complejidad son probablemente distintas. El reto nos lleva entonces a traducir en pruebas matemáticas esta evidencia.

Típicamente, una clase de complejidad cae en dos categorías principales: (a) modelos basados en las máquinas de Turing (TMs) y las maquinas de acceso aleatorio (RAMs), y (b) modelos

basados en circuitos.

3.1.2. Clases Canónicas

Una clase de complejidad se define en función de:

1. Un modelo de cómputo, basados en una máquina, o en circuitos.
2. Un recurso (o una colección de recursos).
3. Una función conocida como cota de complejidad (complexity bound) para el recurso o recursos.

Empezaremos a revisar la importancia de los recursos fundamentales de tiempo y espacio para maquinas de Turing deterministas y no deterministas. Nos concentraremos en las cotas de recursos entre logarítmico y exponencial, porque estas cotas han probado ser las más útiles para entender los problemas que nos encontramos al procesar documentos XML.

Definición 3.3. *Una máquina de Turing determinista M con k cintas (k es un entero y $k \geq 1$), es una cuarteta $M = (Q, \Sigma, \delta, s)$, en la que Q es un conjunto finito de estados, Σ es un conjunto finito de simbolos (se dice que Σ es el alfabeto de M), asumimos que $Q \cap \Sigma = \varnothing$, $s \in Q$ y corresponde al estado inicial; δ es la función parcial de transferencia, $\delta : Q \times \Sigma^k \rightarrow (Q \cup \{alto, \text{“}si\text{”}, \text{“}no\text{”}\}) \times (\Sigma \times \{\leftarrow, \rightarrow, -\})^k$.*

De esta manera podemos ver que δ determina la transición de M hacia el siguiente estado general de todos sus componentes, es decir $\delta(q, \sigma_1, \ldots \sigma_k) = (p, \rho_1, D_1, \ldots, \rho_k, D_k)$ significa que, si DTM esta en el estado q, el cursor de la primera cadena esta leyendo un σ_1, que la segunda lee un σ_2, y así sucesivamente hasta la $k-$ésima cadena, leyendo un σ_k; estas condiciones llevan a M hacia el estado p, el cursor de la primera cadena sobrescribe en esta ρ_1 y se mueve en la dirección indicada por D_1, de la misma manera se realizan las operaciones de sobrescritura y movimiento en el resto de las k cadenas.

Consideramos que Σ contiene los símbolos de “espacio en blanco” $\sqcup$ y de “inicio de cadena” $\triangleright$, cuando este segundo símbolo es detectado por el cursor que explora una cadena ($\sigma_i = \triangleright$), siempre produce dos reacciones en automático: la cadena no se sobrescribe, entonces el símbolo $\triangleright$ nunca se borra ($\rho_i = \triangleright$), y el movimiento es hacia la derecha ($D_i = \rightarrow$), es decir el cursor nunca rebasa el símbolo $\triangleright$ por la izquierda.

Definición 3.4. *Una máquina de Turing no determinista N con k cintas (k es un entero y $k \geq 1$), es una cuarteta $M = (Q, \Sigma, \delta, s)$, en la que Q es un conjunto finito de estados, Σ es un conjunto finito de simbolos, asumimos que $Q \cap \Sigma = \varnothing$, $s \in Q$ y corresponde al estado inicial; δ es una relación de transferencia, $\delta \subset Q \times \Sigma^k \times Q \times \Sigma^k \times \{\leftarrow, \rightarrow, -\}^k$.*

Un elemento $(q, (a_1, a_2, \ldots, a_k), q', (b_1, b_2, \ldots, b_k), (m_1, m_2, \ldots, m_k))$ del conjunto δ constituye una transición. Una descripción instantánea α de una máquina con una cinta es una pareja $\alpha = (q, \overline{w}.x.\overline{w'})$, donde $x \in \Sigma$ y si $\overline{w}x\overline{w'}$ esta escrito en la cinta. Esta noción se generaliza en

caso de máquinas con k cintas, una noción importante es la relación de transición entre dos configuraciones instantáneas: $\alpha \vdash \alpha'$ si existe una transición que permite definir α' a partir de α. En el caso de una máquina no determinista, tratamos con un **árbol de cálculo** definido por las diferentes sucesiones posibles de transición.

Definición 3.5. *Una máquina de Turing M* ***acepta la palabra*** *$x = x_1.x_2.\ldots.x_n$ si existe un estado de aceptación $q_f \in Q$ tal que la máquina alcanza alguno de ellos a partir de la configuración inicial $(q_0, .x_1.x_2.\ldots.x_n)$.*

El lenguaje L ***asociado*** *a una máquina de Turing M, es el conjunto de palabras aceptado por la máquina M. Decimos que M* ***decide*** *L si la máquina de Turing a la entrada de una cadena $x \in \Sigma^*$ es capaz de detenerse en un estado de aceptación, $M(x) =$"si" o de rechazo $M(x) =$"no"; a L se le reconoce como un* ***lenguaje recursivo****; si solamente nos indica cuando lo acepta y en el resto de los casos M no se detiene $M(x) = \nearrow$, se dice que M* ***acepta a*** *L y a este lenguaje se le reconoce como* ***lenguaje recursivo enumerable****.*

Para una cadena de entrada $x \in \Sigma^*$, si tenemos que $T(x) = M^k$ es el número k de transiciones de la máquina determinista M antes de alcanzar un estado de aceptación o de rechazo:

Definición 3.6. *La función de complejidad en tiempo de la máquina M se denota por $T_M : \mathbf{N} \to \mathbf{N}$ y se define como: $T_M(n) = \max\{T(x)||x| = n\}$.*

Si L es un lenguaje sobre el alfabeto Σ y f una función de $\mathbf{N}$ en $\mathbf{N}$, tenemos las siguientes ***clases de tiempo*** *fundamentales:*

L pertenece a la clase $\mathsf{DTIME}[f(n)]$ si existe una máquina de Turing determinista M con k cintas que acepta a L con una complejidad en tiempo $T_M = \mathcal{O}(f(n))$.

L pertenece a la clase $\mathsf{NTIME}[f(n)]$ si existe una máquina de Turing no determinista M con k cintas que acepta a L con una complejidad en tiempo, considerando todos las trayectorias de su árbol de cálculo, $T_M = \mathcal{O}(f(n))$.

Definición 3.7. *La función de complejidad en espacio de una máquina M se denota por $S_M : \mathbf{N} \to \mathbf{N}$ y se define como: $S_M(n) = \max\{S(x)||x| = n\}$.*

Si L es un lenguaje sobre el alfabeto Σ y f una función de $\mathbf{N}$ en $\mathbf{N}$, tenemos las siguientes ***clases de espacio*** *fundamentales:*

L pertenece a la clase $\mathsf{DSPACE}[f(n)]$ si existe una máquina de Turing determinista M con k cintas que acepta a L con una complejidad en espacio $S_M = \mathcal{O}(f(n))$.

L pertenece a la clase $\mathsf{NSPACE}[f(n)]$ si existe una máquina de Turing no determinista M con k cintas que acepta a L con una complejidad en espacio, considerando todos las trayectorias de su árbol de cálculo, el espacio utilizado en las cintas de trabajo tiene una complejidad en espacio $S_M = \mathcal{O}(f(n))$.

Para una máquina de Turing determinada, hablamos de la clase $\mathcal{O}(f(n))$ en tiempo, o $\mathsf{DTIME}(\mathcal{O}(n))$ indicando la clase de lenguajes para los cuales existe una máquina determinista que termina (aceptando o rechazando) en tiempo lineal, en función del tamaño de la entrada. Por extensión decimos que un **problema esta en una clase** C si el lenguaje asociado L está en la clase C.

Definición 3.8. *Las **clases canónicas de complejidad**, con sus diferentes notaciones, son las siguientes:*

LogSpace= $\mathsf{L} = \mathsf{DSPACE}[\log n]$ *(espacio* log *determinista)*

NLogSpace= $\mathsf{NL} = \mathsf{NSPACE}[\log n]$ *(espacio* log *no determinista)*

PTime= $\mathsf{P} = \mathsf{DTIME}[n^{\mathcal{O}(1)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[n^k]$ *(tiempo polinomial)*

NPTime= $\mathsf{NP} = \mathsf{NTIME}[n^{\mathcal{O}(1)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[n^k]$ *(tiempo polinomial no-determinista)*

PSpace= $\mathsf{PSPACE} = \mathsf{DSPACE}[n^{\mathcal{O}(1)}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[n^k]$ *(espacio polinomial)*

ETime= $\mathsf{E} = \mathsf{DTIME}[2^{\mathcal{O}(n)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[k^n]$

NETime= $\mathsf{NE} = \mathsf{NTIME}[2^{\mathcal{O}(n)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[k^n]$

ExpTime= $\mathsf{EXP} = \mathsf{DTIME}[2^{n^{\mathcal{O}(1)}}] = \bigcup_{k\geq 1} \mathsf{DTIME}[2^{n^k}]$ *(tiempo exponencial determinista)*

NExpTime= $\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{\mathcal{O}(1)}}] = \bigcup_{k\geq 1} \mathsf{NTIME}[2^{n^k}]$ *(tiempo exp. no determinista)*

ExpSpace= $\mathsf{EXPSPACE} = \mathsf{DSPACE}[2^{n^{\mathcal{O}(1)}}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[2^{n^k}]$ *(tiempo exponencial)*

Las clases PSPACE *y* EXPSPACE *se definen en términos de la medida de la complejidad* DSPACE.

Por el Teorema de Savitch [Pap94, pag.149], la medida NSPACE con cotas polinomiales también produce PSPACE, y con cota $2^{n^{\mathcal{O}(1)}}$ produce EXPSPACE.

Como una consecuencia de la primera parte de este teorema, $\mathsf{NP} \subseteq \mathsf{EXP}$. Ninguna mejor cota general superior que esta se conoce para lenguajes en NP. Aunque no conocemos si permitimos un nodeterminismo estricto, esto incrementa la clase de lenguajes decidibles en tiempo polinomial, el Teorema de Savitch establece que para clases en espacio, el nodeterminismo no incrementa el poder computacional más allá de una cantidad polinomial.

3.1.3. Complejidad en los Circuitos Lógicos

La máquina de Turing constituye un modelo secuencial de cómputo, es decir efectua una sola instrucción por etapa, como contraparte los circuitos lógicos, también denominados booleanos, constituyen un modelo de cómputo paralelo ya que realizan una cantidad polinomial de operaciones elementales por unidad de tiempo. El conjunto de estas operaciones corresponden a los operadores de la lógica booleana, es decir $\Omega = \{\neg, \vee, \wedge\}$. Una fórmula proposicional

$\psi(x_1, x_2, \ldots, x_n)$ es una fórmula construida a partir de variables booleanas $x_1, \ldots, x_n$ que admite una descomposición bajo la forma de un árol binario en el que sus nodos son estas variables x_i en $\{0, 1\}$ y el árbol entonces representa un circuito lógico que realiza un cómputo **paralelo**, ya que en cada etapa varias compuertas (nodos) realizan operaciones elementales. Los modelos de cómputo permiten calcular entradas de tamaño arbitrario. En el caso de los circuitos, nos referimos a familias de circuitos

$$\mathcal{C} = \{C_1, C_2, \ldots, C_n, \ldots\}$$

El circuito C_n procesa todas las entradas de tamaño n. Recordemos que el grado interior de un nodo v de una gráfica G es la cantidad de aristas (u, v) entrantes en v, y el grado exterior de v es la cantidad de aristas (v, w) salientes de v.

Definición 3.9. *Un **circuito lógico** o **circuito booleano** C_n es una gráfica acíclica, en la que los nodos de grado interior no nulo están etiquetados por los elementos de $\Omega = \{\neg, \vee, \wedge\}$, los nodos de grado interior nulo están etiquetados por las variables x_i para $i = 1, 2, \ldots, n$ denominadas **variables de entrada** Los nodos de grado exterior nulo están etiquetados por las variables $y_1, y_2, \ldots y_m$, denominadas **variables de salida**.*

*Los nodos etiquetados con $\vee$,$\wedge$ y $\neg$ se denominan, compuertas **AND**, **OR** y **NOT** respectivamente.*

*La cantidad de aristas que llegan a una compuerta recibe el nombre de grado interior o **fan-in**, la cantidad de aristas salientes es su grado exterior o **fan-out***

*Los parametros característicos de un circuito son el tamaño (size) y la profundidad (depth): El **tamaño** $S(C)$ de un circuito C es la cantidad de nodos en su gráfica, mientras que la **profundidad** $D(C)$ de un circuito C es la longitud de la trayectoria más larga en su gráfica.*

El fan-in mínimo es 2 para los nodos etiquetados con $\wedge$ ó $\vee$ y 1 para los nodos etiquetados por $\neg$. Su fan-out es 1 como máximo.

Un circuito con n entradas corresponde a una gráfica acíclica con $2n$ nodos fuente, los cuales se etiquetan como:

$$x_1, \neg x_1, \ldots, x_n, \neg x_n.$$

Para cada una de las compuertas g del circuito C con n entradas esta asociado a una función:

$$f : \{0, 1\}^n \to \{0, 1\},$$

definiendo por inducción sobre la longitud de la trayectoria más larga desde una compuerta de entrada hasta g, como sigue: Si g es una compuerta de entrada etiquetada como x_i, entonces:

$$F_g(a_1, \ldots, a_n) = a_i$$

Si g es una compuerta de entrada etiquetada como $\neg x_i$, entonces:

$$F_g(a_1, \ldots, a_n) = 1 - a_i$$

Es decir:

$$F_g(a_1, \ldots, a_n) = f_g(F_{g_1}(a_1, \ldots, a_n), \ldots, F_{g_r}(a_1, \ldots, a_n)),$$

donde r aristas entran a la compuerta g, f_g es la función que etiqueta la compuerta g, y g_i es el origen de la $i-$*ésima* arista que entra a la compuerta g. En particular, un circuito C con n entradas y k compuertas de salida que están ordenados en alguna manera computa una función $F_C : \{0,1\}^n \to \{0,1\}^k$. Si $k = 1$, decimos que el circuito *acepta* cada uno de los bits de la cadena $a_1 \cdots a_n$ para los cuales $F_C(a_1, \ldots, a_n) = 1$, y que el circuito *reconoce* el conjunto de cadenas que acepta. El circuito con 0 entradas, corresponde a un conjunto de compuertas de salida cada una de las cuales esta etiquetada con 1 ó 0.

Una trayectoria en una gráfica acíclica es una serie de aristas adyacentes con la misma orientación. Para estos circuitos podemos generalizar al considerar las funciones definidas por:

$$\wedge_n(x_1, x_2, \ldots, x_n) = (x_1 \wedge x_2 \wedge \ldots \wedge x_n)$$

y

$$\vee_n(x_1, x_2, \ldots, x_n) = (x_1 \vee x_2 \vee \ldots \vee x_n)$$

Un circuito con fan-in n es un circuito en el que los nodos están etiquetados con $\wedge_n$ o $\vee_n$. Por consiguiente estos nodos son de grado interior n en la gráfica asociada al circuito. Los valores de las funciones $\wedge_n$ y $\vee_n$ son aquellas definidas por las fórmulas precedentes.

De esta forma podemos escribir, $C_n^i(x_1, x_2, \ldots, x_n) = y_i$, para notar la función que sobre las entradas $x_i, x_2, \ldots, x_n$ da la $i-esima$ salida del circuito. A un problema de decisión se asocia una familia de circuitos con una sola salida y_1 y podemos interpretar a $y_1 = 1$ como el hecho de aceptar a $\mathbf{x}$, y a $y_1 = 0$, como el resultado de rechazar a $\mathbf{x}$. A un problema de cómputo, asociamos una familia de circuitos con una salida y_1 que definen una familia de funciones f tal que $f(x) = y$ en la que $y = y_1, y_2, \ldots y_m$ si el circuito $C_{|x|}$ tiene exactamente m variables de salida. De igual manera podemos asociar a un circuito una fórmula proposicional sobre las variables $x_1, x_2, \ldots, x_n$, obtenida al describir las salidas del circuito por composición sucesiva. Sea $B_n = \{f : \{0,1\}^n \to \{0,1\}\}$ la clase de funciones booleanas con n variables. Un circuito con una salida define una función $f \in B_n$ particular.

Definición 3.10. *Una función $f \in B_n$ esta definida por un circuito C_n con una salida si cualquiera que sea $x \in \{0,1\}^n$, $f(x) = C_n^1(x)$.*

La complejidad en tamaño y la complejidad en profundidad de una función f es el tamaño y la profundidad, respectivamente, del circuito más pequeño que define f.

Un Lenguaje L admite circuitos de tamaño polinomial si existe un polinomio $p(n)$ y una familia de circuitos $\mathcal{C}$ de una salida tal que para toda x de longitud n, $x \in L$ si y solo si $C_n(x) = 1$ y $S(C_n) \leq p(n)$.

Si un lenguaje L esta en la clase P, entonces L admite circuitos polinomiales.

Supongamos una clase $\mathcal{C}$ de circuitos con tamaño polinomial. Un problema importante es saber como esta definido el circuito C_n para una determinada n. Las clases de circuitos $\mathcal{C} =$

$\{C_1, C_2, \ldots C_n, \ldots\}$ considerando que son tales que la función para toda n describe al circuito C_n como una gráfica etiquetada y calculable en espacio logarítmico.

En una familia de circuitos, donde sus componentes tienen diferente cantidad de entradas, en el caso donde cada circuito de la familia tiene solamente una compuerta de salida, la familia reconoce un lenguaje $L \subseteq \{0,1\}^*$, donde $L \cap \{0,1\}^n$ es el conjunto de cadenas reconocidas por el circuito de la familia con n entradas.($L \cap \{0,1\}^n = \emptyset$ si en la familia no existe tal circuito). En el caso de varias salidas, la familia computa una función $f : \{0,1\}^* \to \{0,1\}^+$, con la propiedad que si $u, v \in \{0,1\}^*$ tienen la misma longitud, entonces $f(u)$ y $f(v)$ tienen también la misma longitud.

Definición 3.11. *Las principales clases de complejidad de los circuitos lógicos son las siguientes:*

AC^0 *es la clase de problemas que se pueden resolver por circuitos con tamaño polinomial y profundidad constante, construido a partir de compuertas AND, OR y NOT con fan-in sin cotas, esta clase corresponde al cómputo con tiempo* $\mathcal{O}(\log^0 n) = \mathcal{O}(1)$ *en una computadora paralela. La clase* $\mathsf{AC}^0(m)$ *o* ACC^0 *corresponde a problemas que se resuelven por circuitos de tamaño polinomial, profundidad constante y que están construidos con compuertas AND,OR,NOT y MODm y fan-in no acotado. Una compuerta MODm a partir de las entradas* $x_1, \ldots, x_n$ *determina si el número de 1's entre estas entradas es un múltiplo de m*

NC^1 *es la clase de problemas que pueden ser resueltos por circuitos con compuertas AND,OR y NOT con fan-in dos y profundidad* $\mathcal{O}(\log n)$.

TC^0 *es la clase de problemas que pueden ser resueltos por circuitos con umbral (threshold) de tamaño polinomial y profundidad constante. Utiliza el tipo de compuerta denominada MAJORITY con un fan-in sin cota.*

NC^0 *es la clase de problemas que pueden ser resueltos por circuitos construidos a partir de compuertas AND,OR y NOT con fan-in igual a dos y profundidad* $\mathcal{O}(1)$.

La clase AC^0 corresponde exactamente a los lenguajes que pueden ser especificados en lógica de primer orden (FO), estos circuitos son suficientemente poderosos para sumar y restar numeros de n-bits.

Los circuitos de la clase NC^1 capturan exactamente la complejidad de un circuito que se requiere para evaluar una fórmula booleana, y reconocer un conjunto regular. Esta clase esta constituida con los lenguajes reconocidos por familias de circuitos $\mathcal{C}_n \geq 1$, con compuertas *AND* y *OR* que en el cual cada circuito tiene una compuerta de salida, y que satisfacen las siguientes condiciones:

- $\mathcal{C}_n$ tiene n entradas.
- Para toda n, cada una de las compuertas *AND* y *OR* tiene un fan-in de dos.
- Existe $k > 0$ tal que para toda n, la profundidad de $\mathcal{C}_n$ es menor a $k \bullet \log_2 n$.
- Existe un polinomio p tal que el tamaño de $\mathcal{C}_n$ es menor que p_n.

En los circuitos de la clase NC^0, cada una de los bits de salida solamente depende de $\mathcal{O}(1)$ bits de entrada del mismo circuito. Esta clase es extremadamente limitada, ya que no pueden computar el *OR* lógico de n bits de entrada, sin embargo esto no impide que su poder computacional este muy cerca de AC^0.

En cuanto a la clase TC^0 podemos hacer notar que captura exactamente la complejidad de la multiplicación, división y ordenación de enteros, también es un buen modelo para la compuerta *MAJORITY* con un fan-in sin cota, utilizada en una red neuronal.

Definición 3.12. *Una familia de circuitos $\mathcal{C} = \{C_1, C_2, \ldots C_n, \ldots\}$ es P-uniforme, denotado como uP si la función g tal que $g(n) = C_n$ es calculable en tiempo polinomial para n representada en notación unaria.*

Una familia de circuitos $\mathcal{C} = \{C_1, C_2, \ldots C_n, \ldots\}$ es L-uniforme si la función g tal que $g(n) = C_n$ es calculable en espacio logarítmico $\mathcal{O}(\log n)$ para n representada en notación unaria.

*L es de complejidad en **tiempo paralelo y tamaño** $f(n)$ si existe una familia de circuitos $\mathcal{C} = \{C_1, C_2, \ldots C_n, \ldots\}$, L-uniforme tal que para n y para toda x de tamaño n: xinL sí y solo si $C_n(x) = 1$, la profundidad y el tamaño respectivamente de C_n es inferior a $k.f(n)$.*

Estas definiciones permiten establecer el resultado clásico referente a que los circuitos *uniformes* de tamaño polinomial son equivalentes en poder de cómputo a una maquina de Turing que utilizan tiempo polinomial, la misma demostración que prueba lo anterior permite mostrar que las clases de estos dos tipos de circuitos uniformes, P-uniforme y L-uniforme, son equivalentes una a otra.

El concepto de ***uniformidad*** en las familias de circuitos, la ilustraremos con NC^1, sabemos que esta clase corresponde a una familia de lenguajes que se reorganizan para algoritmos en paralelo particularmente rápidos. Pero NC^1 contiene lenguages no-recursivos, de la misma manera N sea un conjunto no-recursivo de enteros positivos, y sea $\mathcal{C}_i$ sea el circuito con una compuerta $a_i \vee \neg a_i$, si $i \in N$, y $a_i \wedge \neg a_i$, si $i \notin N$. Esta familia reconocerá el lenguaje no-recursivo $\{w \in \{0,1\}^* : |w| \in N\}$.

Esta situación ocurre ya que tenemos un modelo computacional definido por una secuencia de dispositivos indexados por la longitud de la entrada. Una manera para hacer la teoria de la complejidad de modelos más cercanos a la complejidad comun de una máquina de Turing, o a la teoría de la complejidad para modelos de computadoras con arquitectura paralela, es necesario introducir la condición de uniformidad.

Esto corresponde a un requerimiento que, dada n, existe un algoritmo eficiente para construir el $n-ésimo$ dispositivo en la secuencia. Para que se cumpla esta condición, todo lenguaje en NC^1 es recursivo. Generalmente la condición se formula de tal manera que todo lenguaje en NC^1 sea reconocido en espacio logaritmico. De la misma forma, con una condición de uniformidad apropiada, todo lenguaje reconocido por una familia de circuitos de tamaño polinomial construido a partir de compuertas *AND*, compuertas *OR* y compuertas *NOT*, es reconocible en tiempo polinomial. La mayoría de los lenguajes "naturales" en NC^1 se conoce que se encuentran en NC^1 cuando se les impone esta condición, pero existen algunas excepciones. Por ejemplo, no se conoce como multiplicar n enteros de n-bits utilizando un espacio logarítmico;

la construcción propuesta consistente en una familia de circuitos con profundidad logaritmica para este problema utiliza más espacio.

Las clases de complejidad uniformes corresponden a las familias de circuitos de profundidad $(\log n)^i$ y de tamaño polinomial. Un problema es suceptible de ser procesado en paralelo, es decir **paralelizable**, si el problema es de complejidad en tiempo paralelo $(\log n)^i$, es decir si existe una familia de circuitos de profundidad $(\log n)^i$ y de tamaño polinomial que permite decidirlo.

Definición 3.13. *La clase* uNC^i *es la clase de los lenguajes L tales que existe una familia de circuitos de fan-in 2,* $\mathsf{L}-uniforme$*, de profundidad* $\mathcal{O}(\log^i n)$ *y de talla polinomial, que acepta a L.*

$$\mathsf{uNC} = \bigcup_i \mathsf{NC}^i$$

Los circuitos uNC^i solamente utilizan compuertas binarias. Si las compuertas del circuito tienen un fan-in n, es decir si se utilizan compuertas $\wedge_n$ o $\vee_n$, nos estamos referiendo a circuitos uAC^i.

Definición 3.14. *La clase* uAC^i *es la clase de los lenguajes L tales que existe una familia de circuitos de fan-in n,* $\mathsf{L}-uniforme$*, de profundidad* $\mathcal{O}(\log^i n)$ *y de talla polinomial, que acepta a L.*

$$\mathsf{uAC} = \bigcup_i \mathsf{AC}^i$$

LogCFL es la clase de complejidad que contiene todos los problemas de decisión que se pueden reducir en espacio logarítmico a un lenguaje libre de contexto (context-free language). Esta clase esta situada entre NL y AC^1.

3.1.4. Constructibilidad, Reducibilidad y Completidad

Constructibilidad

Sin la noción de *Constructibilidad* ninguna teoría de complejidad significativa es posible. La primera enseñanza que nos plantea la teoría de la complejidad (en contra de lo que la intuición nos indica), dice "Si tiene más recursos, no puede hacer más".

Teorema 3.1. ***Teorema de la Brecha (Gap)***
Existe una cota de tiempo computable $t(n)$ *tal que* $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[2^{2^{t(n)}}]$.

Este teorema nos dice que existe una brecha vacía entre el tiempo $t(n)$ y el tiempo doblemente exponencial más grande que $t(n)$, en el sentido que cualquier cosa que pueda ser computada en una cota de tiempo más grande puede también ser computada en la cota de tiempo más pequeña. Esto es, aún con mucho más tiempo, no podemos computar más. Esta brecha se puede hacer más grande que la doble exponencial; para cualquier r computable, se tiene una cota de tiempo computable t tal que $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[r(t(n))]$. Proposiciones análogas se cumplen para las medidas NTIME, DSPACE, y NSPACE.

Efectivamente, la prueba del Teorema de la Brecha muestra que se puede definir una cota de tiempo $t(n)$ tal que ninguna máquina tiene un tiempo de ejecución que este entre $t(n)$ y $2^{2^{t(n)}}$. Esto nos indica la necesidad de formular solamente aquellas cotas en tiempo que realmente describan la complejidad de una determinada máquina.

Definición 3.15. *Una función $t(n)$ es **constructible-en-tiempo** si existe una máquina de Turing determinista que se detenga después de exáctamente $t(n)$ pasos, para cualquier entrada de longitud n.*

*Una función $s(n)$ es **constructible-en-espacio** si existe una máquina de Turing determinista que use exactamente $s(n)$ celdas de cintas de trabajo para cualquier entrada de longitud n.*

Por ejemplo, $t(n) = n + 1$ es constructible-en-tiempo, Por otra parte si $t_1(n)$ y $t_2(n)$ son constructibles-en-tiempo, entonces también lo son las funciones $t_1 + t_2$, $t_1 t_2$, $t_1^{t_2}$, y c^{t_1} para todo entero $c > 1$. En consecuencia, si $p(n)$ es un polinomio, entonces $p(n) = \Theta(t(n))$ para alguna función $t(n)$ polinomial constructible-en-tiempo. De la misma manera, $s(n) = \log n$ es constructible-en-espacio, si $s_1(n)$ y $s_2(n)$ son constructibles-en-espacio, entonces también las funciones $s_1 + s_2$, $s_1 s_2$, $s_1^{s_2}$ y c^{s_1} para todo entero $c > 1$. Muchas funciones comunes son constructibles-en-espacio: v. gr.: $n \log n$, n^3, 2^n, $n!$.

La constructibilidad contribuye a eliminar una elección arbitraria en la definición de las clases básicas de tiempo y espacio. Para funciones generales en tiempo t, las clases $\mathsf{DTIME}[t]$ y $\mathsf{NTIME}[t]$ pueden variar en función de las máquinas requeridas para detenerse dentro de t etapas en todas las trayectorias de cómputo, o solamente en aquellas trayectorias con aceptación. Si t es constructible-en-tiempo y s es constructible-en-espacio, sin embargo con estos antecedentes, $\mathsf{DTIME}[t]$, $\mathsf{NTIME}[t]$, $\mathsf{DSPACE}[s]$, y $\mathsf{NSPACE}[s]$ pueden definirse sin pérdida de generalidad en términos de máquinas de Turing que siempre se detienen.

Como regla general, cualquier función $t(n) \geq n + 1$ y cualquier función $s(n) \geq \log n$ en la que se este interesado como una cota de tiempo o espacio, es constructible -en-tiempo o -en espacio, respectivamente.

El Teorema de la Brecha no es el único caso en que la intuición acerca de la complejidad nos lleva a conclusiones falsas. La mayor parte de los informáticos también espera que un objetivo del diseño de algoritmos debería ser llegar a un algoritmo óptimo para un determinado problema. En algunos casos, sin embargo, ningún algoritmo esta remotamente cerca del óptimo.

Teorema 3.2. ***Teorema de la Aceleración Incrementada***
Existe un lenguaje decidible A que para toda máquina M que decide A, con tiempo de ejecución $u(n)$, existe otra máquina M' que decide A mucho más rápido: su tiempo de ejecución $t(n)$ satisface $2^{2^{t(n)}} \leq u(n)$ para todo n finito.

Esta afirmación, también, se cumple para cualquier función computable $r(t)$ en lugar de 2^{2^t}. Nos dice intuitivamente que el programa M' ejecutándose en una PC IBM antigua es mejor que el programa M ejecutándose en el hardware más rápido que existe actualmente, por consiguiente no tenemos para A un mejor algoritmo, ni una función de complejidad en tiempo bien definida. A diferencia del caso del Teorema de la Brecha, el fenómeno de la aceleración

puede ser válido para lenguajes y cotas en tiempo de interés, por ejemplo, un problema de complejidad en tiempo acotada por $t(n) = n^{\log n}$, ubicado exactamente arriba del tiempo polinomial, puede tener aceleración arbitraria polinomial, esto es, puede tener algoritmos de complejidad en tiempo $t(n)^{1/k}$ para toda $k > 0$.

Una implicación del Teorema de la Aceleración es que las complejidades de algunos problemas necesitan constreñirse entre las cotas superior e inferior, Para todo lenguaje A se tiene una función computable t_0 tal que para toda función t constructible-en-tiempo, existe alguna máquina que acepta A dentro del tiempo t si y sólo si $t = \Omega(t_0)$. Sin embargo, como consecuencia tenemos que t_0, por si mismo, no puede ser constructible-en-tiempo.

Reducibilidad

Una manera en la que podemos obtener una solución para un problema es a través del concepto de reducibilidad entre problemas, este procedimiento lo conocemos en la vida diaria: para resolver un problema que no sabemos resolver fácilmente, lo trasladamos, posiblemente con otra representación, a un escenario que nos es más favorable. Si podemos reducir un problema a otro, podemos usar un algoritmo para el segundo problema y obtener una solución, y entonces transformar esta respuesta en una solución para el primer problema. Si podemos hacer las transformaciones en tiempo polinomial y el segundo problema puede resolverse en tiempo polinomial, sabemos que nuestro nuevo problema también tiene una solución en tiempo polinomial.

Veamos un ejemplo acerca de esto para clarificar el proceso. Nuestro primer problema regresara "si" dentro de un grupo de variables booleanas si al menos una de ellas tiene el valor de verdadero, y regresará "no" si todas son falsas. El segundo problema consiste en encontrar el valor más grande en una lista de enteros. Debemos ser capaces de ver una solución clara y fácil para cada uno de estos problemas, pero el objetivo de este ejemplo es suponer que tenemos una solución para el problema del entero más grande pero no par el problema de las variables booleanas. Podemos resolver el problema de la variable booleana reduciéndolo al problema del entero más grande. Empezamos tomando una instancia del problema de la variable boleana y escribir un algoritmo de conversión que asignará, para cada variable booleana la siguiente entrada en la lista el valor de 0 si la variable booleana es falsa y un valor de 1 si la variable es verdadera. A continuación utilizamos nuestro algoritmo para encontrar el valor más grande en la lista. Veremos que la respuesta regresa dentro de la solución al problema de la variable booleana al responder con "si" en caso que el valor más grande es 1 y regresar no si el valor más grande es 0.

El problema del valor más grande dentro de una lista puede ejecutarse en tiempo lineal, y vemos que nuestra reducción también se puede realizar en tiempo lineal; por consiguiente, el problema de la variable booleana debe también tener solución en tiempo lineal.

De esta manera podemos decir que un problema B se reduce al problema A si existe una transformación R la cual, para cada entrada x de B, produce una entrada equivalente $R(x)$ de A; en este planteamiento por "equivalente" queremos decir que la respuesta a $R(x)$ considerada como una entrada para A, "si" o "no", corresponde a una respuesta correcta para x, la cual

constituye una entrada de B. En otras palabras, para resolver B con una entrada x solamente tenemos que computar $R(x)$ y resolver A con estos antecedentes.

Otro ejemplo que podemos plantear, es la reducción del problema de validez de una fórmula de la lógica de proposiciones al problema de insatisfacibilidad: si queremos determinar si una fórmula ϕ es lógicamente válida, consideramos la transformación hacia $\neg\phi$, y determinamos si esta última es o no satisfacible, recordemos que:

- ϕ es lógicamente válida si y sólo si $\neg\phi$ es insatisfacible,
- el paso de ϕ a $\neg\phi$ es computable por un algoritmo general.

Con base en esto se puede generalizar el concepto de reducibilidad:

Definición 3.16. *Sean $L_1 \subseteq \Sigma_1^*$ y $L_2 \subseteq \Sigma_2^*$. Decimos que L_1 es **reducible** a L_2, y escribimos $L_1 \leq L_2$, si existe una función f de Σ_1^* en Σ_2^*, tal que:*

- *para todo $w \in \Sigma_1^* : w \in L_1 \Leftrightarrow f(w) \in L_2$*
- *$f : \Sigma_1^* \to \Sigma_2^*$ es computable*

Podemos observar a partir de esta definición que, si $L_1 \leq L_2$, obtenemos de manera inmediata los siguientes hechos:

1. L_2 decidible $\Rightarrow$ L_1 decidible.
2. L_1 indecidible $\Rightarrow$ L_2 indecidible.

En efecto, si, como en 1., L_2 es decidible, entonces, para obtener un algoritmo de decisión para L_1, basta con computar, para cada $w \in \Sigma_1^*$, su imagen $f(w)$ y decidir si $f(w) \in Ł_2$. Por otra parte, si como en 2., L_1 es indecidible , entonces el problema más general L_2, en el cual estamos llevando a L_1 de manera efectiva (computable), tampoco puede ser decidible.

Como se ha estado prefigurando el proceso de reducción, para que realmente sea significativa, esta debe involucrar al cómputo mas débil posible, por consiguiente podemos establecer la siguiente definición:

Definición 3.17. *La reducción acotada en espacio* $\log n$ *representa una* reducción eficiente*; es decir un lenguaje L_1 es reducible eficientemente al lenguaje L_2 si existe una función R computable, de cadena a cadena, por una máquina de Turing determinista en espacio $\mathcal{O}(\log n)$ de tal manera que para todas las entradas x lo siguiente es verdadero: $x \in L_1$ si y solamente si $R(x) \in L_2$. R representa una **reducción eficiente** de L_1 a L_2.*

La relevancia que representa el concepto de reducibilidad al comparar clases de complejidad, es importante tener presentes los procesos de reducción que nos llevan a *algoritmos contiempo polinómico.*

Teorema 3.3. *Si R es una reducción computable por una máquina de Turing M, entonces para todas las entradas x, M se detiene después de un número polinomial de pasos.*

Prueba: Se tienen $\mathcal{O}(nc^{\log n})$ posibles configuraciones para M con entrada x, con $n = |x|$. Ya que la máquina es determinista, ninguna configuración se puede repetir en el proceso de cómputo (debido a que tal repetición representa que la máuina no se detiene). Por lo tanto, el cómputo tiene una longitud máxima de $\mathcal{O}(n^k)$ para una determinada k

Definición 3.18. *Una máquina de Turing con oráculo es una máquina de Turing equipada con una cinta extra denominada cinta de consulta. Sea M una máquina de Turing con oráculo y B cualquier consulta de tipo booleano. Podemos escribir M^B para denotar la máquina de Turing con oráculo M para un conjunto B. M^B puede escribir sobre su cinta de consulta como en cualquier cinta. En cualquier tiempo, M^B puede entrar en el "estado de consulta". Si asumimos que la cadena binaria w que codifica a la estructura $\mathcal{A}$:*

$$w = bin(\mathcal{A})$$

esta escrita en la cinta de consulta en el momento en que M^B entra en el estado de consulta. En la siguiente etapa, aparecerá 1 sobre la cinta de consulta si $\mathcal{A} \in B$, y 0 en el otro caso. Por consiguiente M^B puede responder en tiempo lineal a cualquier pregunta de membresía ¿$\mathcal{A}$ satisfase B?: el tiempo durante el que se copia a la cinta de consulta la cadena w.

Si tenemos dos consultas A y B y una clase de complejidad C, decimos que A es C-Turing reducible *a B si y sólo si existe una máquina de Turing con oráculo M tal que M^B ejecuta en complejidad de clase C y $L(M^B) = A$. Esto se denota por $A \leq_{Turing-\mathsf{C}} B$*

La noción de reducción se introdujo desde el origen de la teoría de la computabilidad para clasificar los conjuntos desde el punto de vista de la recursividad. Las principales reducciones, la m-reducción y la reducción de Turing entre dos lenguajes A y B, se generalizan en tiempo polinomial.

Definición 3.19. *A es* ***m-reducible en tiempo polinomial*** *en B (cuya notación es $A \leq_P B$), si existe una función f, calculable en tiempo polinomial, tal que para toda entrada x:*

$$x \in A \quad si\ y\ sólo\ si\ f(x) \in B$$

En estos términos, en la m-reducción, para resolver el problema ¿$x \in A$?, es suficiente resolver el problema ¿$f(x) \in B$?.

Definición 3.20. *A es* ***Turing-reducible en tiempo polinomial*** *a B, $B(A \leq_{Turing-P} B)$ si existe una máquina de Turing M con B como oráculo tal que para toda entrada x:*

$$x \in A \quad ssi\ M\ acepta\ a\ x$$

Podemos, entonces, notar que la m-reducción es más restrictiva que la reducción de Turing: en el primer caso llamamos una vez al oráculo B, mientras que en el segundo caso se llama al oráculo B una cantidad polinomial de veces. Existen también otras reducciones que se basan ya sea en fórmulas booleanas (reducción por tabla de verdad), o bien en fórmulas de la FO (reducción de primer orden) con notación $\leq_{FO}$.

Completidad

Cuando C es una clase de complejidad débil como FO o LogSpace, el significado de $A \leq_{\mathsf{C}} B$ es que la complejidad del problema A es menor o igual que la complejidad del problema B. El significado intuitivo de que A sea completo en C es que A es un problema (o una consulta) más difícil en C y de hecho que todo problema (o consulta) en C puede ser replanteada como una instancia de A; de forma más precisa tenemos:

Definición 3.21. *Completidad en una clase de complejidad: Sea A una consulta booleana, C una clase de complejidad y $\leq_r$, una relación de reducibilidad. Decimos que A es* ***completo dentro de la clase C a través de*** *$\leq_r$, si y sólo si:*

1. *$A \in \mathsf{C}$, y,*
2. *para todo $B \in \mathsf{C}$, $B \leq_r A$.*

Por razones que todavía no se entienden bien, observan Neil Immerman y Christos Papadimitriou, de manera natural los problemas en las clases de complejidad importantes, como P, NP, y NL, tienden a ser completos. La complitud se definió originalmente a través de reducciones como la polinomial en tiempo ($\leq_{\mathsf{P}}$) y la logarítmica en espacio ($\leq_{\mathsf{L}}$). sin embargo los problemas completos vía estas reducciones tienden a permanecer completos vía la FO ($\leq_{\mathsf{FO}}$).

3.2. Lógica Computacional

El propósito de la lógica en la Ciencia de la Computación es desarrollar lenguajes para modelar las situaciones que nos encontramos en sus diferentes áreas de estudio, y de esta manera poder razonar formalmente acerca de ellos. Este tratamiento nos permite trabajar con argumentos válidos posibilitándonos a defenderlos rigurosamente, o bien que puedan ejecutarse en una máquina.

En función de la naturaleza y tipo del razonamiento que se pretende modelar, en particular su requerimiento de expresividad, se aplican lógicas ad-hoc a una determinada área de la ciencia computacional. Generalmente una lógica resulta ser extensión de otra, pero habrá pares de ellas que son incompatibles, o bien se llegue a la integración de algunas de éstas lógicas.

La lógica elemental o de proposiciones, misma que en forma de lógica booleana formaliza el manejo de los circuitos lógicos, tiene sus limitaciones de expresividad por lo que es necesario extenderla, para pasar a la lógica de predicados o de primer orden (FO), en la cual se incluyen cuantificadores ("para todos" y "existe alguno") sobre elementos de un dominio y la posibilidad de referirnos a propiedades de estos elementos. Al seguir con el proceso de extensión de la FO, por requerimiento de una mayor expresividad, tenemos la lógica de segundo orden (SO), que esta dotada con cuantificadores sobre las propiedades de los elementos, en particular con algunas restricciones sobre la aridad de sus elementos como la SOM que trata exclusivamente con monoides. Las lógicas modales, el caso clásico es el de las modalidades "necesario" y "posible", en estas lógicas encontramos la lógica temporal, que abarca las modalidades de

tiempo “para siempre”, “próximo”, “hasta que”, “finalmente”, al utilizarlas para formalizar los algoritmos de la verificación de modelos (model checking) por ejemplo, nos lleva en particular a la lineal (LTL) y a la ramificada (CTL).

La formulación y desarrollo de una lógica incluye los siguientes elementos:

1. Sintaxis del Lenguaje: Determina el tipo de proposiciones formales que maneja la lógica. Esto se establece a nivel simbólico, dando las reglas sintácticas que permiten reconocer y construir proposiciones admisibles.

2. Semántica: Define de manera precisa el significado de los constituyentes del lenguaje formal, la noción de verdad, y la noción de consecuencia lógica. Esto se logra recurriendo a mundos (contextos, realidades, ...), donde se interpreta los símbolos del lenguaje y otras construcciones sintácticas. Así mismo define la noción de *consecuencia lógica* y de *que se concluye de que*.

3. Sistema deductivo: Dada la naturaleza formal, simbólica, de las proposiciones, se requiere capturar las nociones semánticas de una manera sintáctica formal; esto permite hacer deducciones y determinar conclusiones admisibles a través de la manipulación sintáctica de las proposiciones aceptadas en una determinada base de conocimientos.

4. Correspondencia entre sintaxis y semántica: Establece formalmente la correspondencia exacta entre las formulaciones semántica y sintáctica de la noción de consecuencia lógica. Los resultados de esta comparación nos permiten evaluar la Corrección (Soundness) y la Completidad (Completeness) del sistema deductivo formal; la primera propiedad no permite deducir conclusiones indeseadas, mientras que la segunda califica que sea lo suficientemente poderoso para derivar todas las conclusiones semánticamente admisibles.

3.2.1. Lógica de Proposiciones

Para formalizar el lenguaje de la lógica de proposiciones L, partimos estableciendo que su sintaxis tiene como base un alfabeto formado por:

1. Conectivos lógicos: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

2. Símbolos de puntuación:), (

3. Un conjunto finito P de variables proposicionales, por ejemplo $P = \{p, q, r, s\}$

El lenguaje $L(P)$ construido a partir de este alfabeto, esta constituido por el conjunto de formulas bien formadas (fbf) que están definidas (en notación BNF), como sigue:

$$\phi ::= p|(\neg\phi)|(\phi \wedge \phi)|(\phi \vee \phi)|(\phi \rightarrow \phi)$$

donde p es cualquier proposición atómica elemento de P y toda ϕ en el lado izquierdo de $::=$ representa a cualquier fbf ya construida.

Una fórmula ϕ puede comprobarse que es una fbf, representándola por su árbol de partición, por ejemplo para:

$$(((p \wedge q) \vee p) \rightarrow (p \vee (\neg q)))$$

Si p y q son elementos de P podemos formar una estructura en forma de árbol para representar la fbf. Al tener en las hojas del árbol sola y exclusivamente proposiciones atómicas, se confirma que se trata de una fbf. Otra manera de confirmar la buena formación de una fórmula es que podamos sintácticamente listar, a partir de sus proposiciones atómicas, las subfórmulas de que esta compuesta como fbf, por ejemplo, para la fórmula anterior tenemos:

p
q
$(p \wedge q)$
$((p \wedge q) \vee p)$
$(\neg q)$
$(p \vee (\neg q))$
$(((p \wedge q) \vee p) \rightarrow (p \vee (\neg q)))$.

Definición 3.22. *Una **asignación de verdad** o **valuación** es una función σ que mapea del conjunto finito P de proposiciones atómicas al conjunto $\{0,1\}$: $\sigma : P \rightarrow \{0,1\}$.*

Consideremos un lenguaje de proposiciones y una asignación $\sigma : P \rightarrow \{0,1\}$, Definimos $\sigma : L(P) \rightarrow \{0,1\}$ de la misma manera: dada una fórmula arbitraria ϕ de $L(P)$, el valor de verdad de $\sigma(\phi)$ es:

Caso Básico: Si $\phi \in P$ (fórmula atómica), entonces $\sigma(\phi)$ es el valor $\sigma(\phi)$ que ϕ tenía originalmente como elemento del conjunto P.

Pasos Inductivos:

Si ϕ es $\neg\psi$), entonces $\sigma(\phi) = 1 - \sigma(\psi)$.

Si ϕ es $\sigma(\psi \vee \varphi)$, entonces $\sigma(\phi) = \text{máx}\{\sigma(\psi), \sigma(\varphi)\}$.

Si ϕ es $\sigma(\psi \wedge \varphi)$, entonces $\sigma(\phi) = \text{mín}\{\sigma(\psi), \sigma(\varphi)\}$.

Si ϕ es $\sigma(\psi \rightarrow \varphi)$, entonces $\sigma(\phi) = 0$ si $\sigma(\psi) = 1$ y $\sigma(\varphi) = 0$. En caso contrario, toma el valor 1.

Si ϕ es $\sigma(\psi \leftrightarrow \varphi)$, entonces $\sigma(\phi) = 1$ si $\sigma(\psi) = \sigma(\varphi)$. En caso contrario, toma el valor 0.

Esta definición se puede representar a través de *tablas de verdad*, o bién los conectivos lógicos pueden ser vistos como funciones que envían valores (o pares de valores) de verdad en valores de verdad:

La negación: $f_\neg : 0,1 \rightarrow 0,1$ tal que $0 \mapsto 1$, $1 \mapsto 0$.

La conjunción: $f_\wedge : (\{0,1\})^2 \rightarrow 0,1$ tal que $(0,0) \mapsto 0$, $(0,1) \mapsto 0$, $(1,0) \mapsto 0$, $(1,1) \mapsto 1$.

La disyunción: $f_\vee : (\{0,1\})^2 \rightarrow 0,1$ tal que $(0,0) \mapsto 0$, $(0,1) \mapsto 1$, $(1,0) \mapsto 1$, $(1,1) \mapsto 1$.

La condicional: $f_{\rightarrow} : (\{0,1\})^2 \rightarrow \{0,1\}$ tal que $(0,0) \mapsto 1$, $(0,1) \mapsto 1$, $(1,0) \mapsto 0$, $(1,1) \mapsto 1$.

La bicondicional: $f_{\leftrightarrow} : (\{0,1\})^2 \rightarrow \{0,1\}$ tal que $(0,0) \mapsto 1$, $(0,1) \mapsto 0$, $(1,0) \mapsto 0$, $(1,1) \mapsto 1$.

Notemos que de $(\{0,1\})^2 \rightarrow \{0,1\}$ podemos tener no solamente estas 5 funciones, sino hasta 2^4, que dan lugar a un total de 16 posibles conectivos distintos de dos argumentos binarios con resultado binario, como sigue:

p	q	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
0	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

El significado de estas funciones se explica en la siguiente tabla.

f_1	Es una función constante, se trata de una tautología, $\top$.(top o cima)
f_2	Es la disyunción de p y q, $(p \vee q)$.
f_3	Es el condicional recíproco de p y q, $(p \leftarrow q)$.
f_4	Es un operador de proyección, su valor depende de p, $f_4(p,q) = p$.
f_5	Es el condicional de p y q, $(p \rightarrow q)$.
f_6	Es un operador de proyección, su valor depende de q, $f_6(p,q) = q$.
f_7	Es la equivalencia de p y q, $(p \leftrightarrow q)$.
f_8	Es la conjunción de p y q, $(p \wedge q)$.
f_9	Es la función de Sheffer o barra (stroke) de Sheffer de p y q, $(p\|q)$ cuya notación se lee: p es incompatible con q. También se le denomina función NAND ya que equivale a la negación de la conjunción.
f_{10}	Es la disyunción exclusiva (o no equivalencia) de p y q, $(p \oplus q)$.
f_{11}	Es la negación del operador de proyección cuyo valor depende de p,$f_{10}(p,q) = (\neg p))$.
f_{12}	Es la negación del condicional de p y q, $\neg(p \rightarrow q)$.
f_{13}	Es la negación del operador de proyección cuyo valor depende de q, $f_{13}(p,q) = (\neg q))$.
f_{14}	Es la negación del condicional recíproco de p y q, $(\neg(p \leftarrow q))$.
f_{15}	Es la función de Peirce o barra (stroke) de Peirce de p y q, $(p \downarrow q)$ cuya notación se lee: ni p ni q. También se le denomina función NOR ya que equivale a la negación de la disyunción.
f_{16}	Es una función constante, se trata de una contradicción, $\bot$. (bottom)

Algunos de éstos conectivos resultan redundantes en el sentido que pueden ser definidos a partir de un subconjunto de conectivos completo, es decir por medio de la composición de funciones. Por ejemplo los conectivos lógicos del alfabeto del lenguaje de proposiciones se pueden definir en términos de $\neg$ y $\vee$, decimos que el conjunto de conectivos $\{\neg, \vee\}$ es *funcionalmente completo.*

La pregunta que esperamos responder a través de la semántica para los lenguajes de proposiciones es ¿Cuándo una fórmula bien formada es válida?

Definición 3.23. *Decimos que una fórmula ϕ de $L(P)$ es* ***válida*** *siempre y cuando sea verdadera en todos los casos de su valuación, es decir, si y sólo si,* ***para toda*** *valuación $\sigma : P \rightarrow \{0,1\}$; $\sigma(\phi) = 1$*

A las fórmulas válidas también se le llama *tautologías*.

Definición 3.24. *Decimos que una fórmula ϕ de $L(P)$ es* ***satisfactible*** *siempre y cuando exista una valuación $\sigma : P \to \{0, 1\}$ tal que $\sigma(\phi) = 1$*

Cuando una fórmula no es satisfactible, decimos que es *insatisfactible*, es decir ninguna asignación la hace verdadera, siempre es falsa. También decimos que es una *contradicción*.

Con respecto a estos conceptos tenemos los siguientes hechos:

1. ϕ es satisfactible si y sólo si $\neg\phi$ no es válida.
2. ϕ es insatisfactible si y sólo si $\neg\phi$ es válida.
3. ϕ es válida si y sólo si $\neg\phi$ es insatisfactible.

Saber si una fbf pertenece a una de estas categorías: válida, satisfacible o insatisfacible es *decidible*, es decir, existe un procedimiento mecánico para determinar cuáles son los elementos que pertenecen y cuáles no a uno de estos conjuntos: la forma de hacerlo consiste simplemente en calcular la tabla de verdad de la fórmula en cuestión.

Definición 3.25. *Dadas dos fórmulas, ϕ y φ, se dice que son* ***lógicamente equivalentes****, denotado $\phi \Leftrightarrow \varphi$, si y sólo si la bicondicional de ambas es una tautología.*

Para extender los conceptos ya revisados a un conjunto de fórmulas Σ, basta definir qué significa que una valuación σ hace verdadero, o satisface a Σ. Esto ocurre siempre y cuando σ hace verdadera a cada una de las fórmulas del conjunto. Esto se denota así: $\sigma \models \Sigma$, y decimos que σ es un *modelo*. Entonces definimos:

$$\sigma \models \Sigma ::\Leftrightarrow \text{para toda } \phi \in \Sigma : \sigma \models \Sigma \quad \sigma(phi) = 1$$

Nota: $::\Leftrightarrow$ significa, como en la notación BNF $::=$, que lo que esta a la izquierda de $::\Leftrightarrow$ está definido en términos de lo que está a la derecha del símbolo correspondiente. Toda ésta notación esta en el metalenguaje.

Una forma alternativa de definir la noción de satisfacción de un conjunto de fórmulas pasa por la extensión del concepto de valor de verdad de una fórmula al valor de verdad del conjunto:

$$\sigma(\Sigma) := \text{mín}\{\sigma(\phi) : \phi \in \Sigma\}.$$

Entonces podemos definir $\sigma \models \Sigma \quad ::\Leftrightarrow \quad \sigma(\Sigma) = 1$.

Notemos que si Σ es un conjunto finito de fórmulas, entonces $\sigma \models \Sigma$ siempre y cuando σ haga verdadera a la conjunción de las fórmulas de Σ. Tal como en el caso de una fórmula aislada, Σ puede ser válido, satisfacible o insatisfacible, en función de si toda, alguna o ninguna asignación satisface a Σ, respectivamente; se entiende que P en $L(P)$ es un conjunto finito de proposiciones, y que por consiguiente las asignaciones son compatibles con el lenguaje, es

decir, que son funciones de P en $\{0,1\}$. Un conjunto de fórmulas es insatisfacible también se llama *inconsistente*. Por ejemplo el conjunto de fórmulas:

$$\Sigma = \{(p \wedge q), (\neg q \vee p)\}$$

es satisfacible, ya que para la asignación:

$\sigma_0 : p \mapsto 1, q \mapsto 1$ se tiene: $\sigma_0 \models \Sigma$.

Sin embargo, Σ no es válido, pues la asignación:

$\sigma_1 : p \mapsto 1, q \mapsto 0$ no satisface Σ.

Esto último lo escribimos así: $\sigma_1 \not\models \Sigma$.

Definición 3.26. *Una* ***forma argumentativa*** *es una proceso que permite eventualmente* inferir *una fórmula ψ, llamada* conclusión*, a partir de una sucesión finita de fórmulas proposicionales $\phi_1, \phi_2, \phi_3, \ldots, \phi_n$ denominadas* premisas*, esto lo denotamos por:*

$$\phi_1, \phi_2, \phi_3, \ldots, \phi_n \quad \vdash \psi$$

La forma o secuencia argumentativa es válida *si existe una valuación tal que $\phi_1, \phi_2, \phi_3, \ldots, \phi_n$ toman el valor de 1 y ψ toma el valor de 1. En caso contrario la forma argumentativa es inválida.*

A continuación precisamos el proceso que caracteriza aquellas proposiciones formales que son consecuencia de (son implicadas por) un conjunto de proposiciones que representan cierto conocimiento. El definir este concepto es una de las tareas principales de cualquier lógica.

Definición 3.27. *Dado un conjunto de fórmulas proposicionales $\Sigma = \{\phi_1, \phi_2, \phi_3, \ldots, \phi_n\}$ de $L(P)$ y ψ una fórmula de $L(P)$. ψ es una* ***consecuencia lógica*** *del conjunto Σ si y sólo si, para toda valuación σ, tal que $\sigma(\phi_i) = 1$ para toda $\phi_i \in \Sigma$, entonces $\sigma(\psi) = 1$.*

$$\sigma : P \to \{0,1\} : \quad \sigma \models \quad \text{tal que}$$

En contraste con este proceso, definimos una nueva relación:

$$\phi_1, \phi_2, \phi_3, \ldots, \phi_n \quad \models \psi$$

El proceso semántico que representa consiste en encontrar los "valores de verdad" de las fórmulas atómicas en las premisas y en la conclusión, determinando como opera el significado de los conectivos lógicos sobre estos valores de verdad.

3.2.2. Lógica de Predicados de Primer Orden o FO (First Order logic)

La lógica de proposiciones solamente permite describir construcciones de lenguajes extremadamente simples: por ejemplo las operaciones booleanas sobre las proposiciones. Esto es insuficiente para representar los procedimientos de los lenguajes utilizados en la computación real,

la lingüística, las matemáticas o para formalizar los fragmentos significativos del razonamiento común.

La *lógica de primer orden* se puede revisar dentro del ámbito de un mismo Lenguaje L, en el que se dan los ejemplos de términos y de fórmulas. Las definiciones del conjunto de *términos* y *fórmulas* son llamadas al lenguaje que incluye un símbolo de predicado binario R, un símbolo de función unaria s, dos símbolos de funciones binarias + y . así como dos símbolos constantes c y d. Utilizamos notación aritmética y las reglas de prioridad habituales para los operadores . y +.

Un lenguaje de la lógica de primer orden para predicados queda esencialmente determinado por un conjunto de símbolos iniciales, S, elegido dependiendo de lo que queramos describir, tal como en el lenguaje de proposiciones cuando elegimos las variables proposicionales iniciales. El conjunto de símbolos o vocabulario S será de la forma :

$$S = \{P, \ldots, f, \ldots, c, \ldots\},$$

donde:

$P, \ldots$ son los símbolos para denotar predicados.

$f, \ldots$ son símbolos para denotar funciones (operaciones).

$c, \ldots$ son nombres o constantes para denotar individuos.

Para construir el lenguaje completo asociado a S, necesitamos, aparte del alfabeto inicial S, los símbolos lógicos y de separación que están en todo lenguaje de la FO:

1. $\neg, \quad \wedge, \quad \vee, \quad \rightarrow, \quad \leftrightarrow$ (conectivos lógicos)
2. $=$ (símbolo para un predicado binario especial, la igualdad)
3. $\exists, \quad \forall$ (cuantificador existencial y universal)
4. $x_1, x_2, x_3, \ldots, x_n$ (una lista finita de varibles)
5.), ((símbolos de puntuación)

Los tres primeros tipos de símbolos corresponden a los símbolos lógicos, en el sentido que en un contexto semántico, tienen una interpretación fija, dada por la "lógica" en contraste los símbolos en S, que tienen interpretación variable según un contexto o dominio. Las variables se refieren a individuos del dominio donde se realiza una interpretación del lenguaje. En un lenguaje de la FO solamente se hace cuantificación sobre los individuos, las lógicas de orden mayor tienen la capacidad de cuantificar también sobre las propiedades de los individuos.

Podemos mencionar como ilustración el caso de un sistema de base de datos (DB) relacional que utiliza un modelo ad-hoc que almacena tablas y relaciones que son consultadas (queried) mediante una lógica basada en un lenguaje, el *cálculo relacional*, que precisamente tiene el poder de un cálculo de predicados de primer orden. Supongamos que tenemos una base de datos de una compañía, y una de sus relaciones es la de Reporta_A: se tienen almacenadas parejas

(x, y), en la que x es un empleado, y y es su jefe inmediato. Las jerarquías organizacionales tienden a complicarse dando como resultado una administración con muchos niveles, en este contexto podriamos estar interesados no en el mando inmediato sino en el jefe de un jefe inmediato de un empleado. En el lenguaje SQL, esta consulta se puede plantear como sigue:

```
select R1.empleado, R2.jefe
from Reporta_A R1, Reporta_A R2
where R1.jefe=R2.empleado
```

Esto representa solamente una forma diferente de expresar lo mismo que la siguiente fórmula de la lógica FO:

$$\varphi(x, y) \equiv \exists z(Reporta\_A(x, z) \wedge Reporta\_A(z, y)).$$

Continuando en el mismo sentido jerárquico, podemos preguntar por el jefe del jefe del jefe inmediato de un empleado:

$$\exists z_1 \exists z_2(Reporta\_A(x, z_1) \wedge Reporta\_A(z_1, z_2) \wedge Reporta\_A(z_2, y)$$

En este ejemplo, tenemos más que un simple conjunto de individuos, tenemos, además una *estructura* en ese conjunto que está indicada a través de distintos atributos (relaciones) aplicables a los individuos; en este sentido una BD relacional no es diferente en contexto de una estructura algebraica como la de los números reales.

La FO permite construir distintos lenguajes formales, dependiendo del conjunto de símbolos básicos con que se inicie. Al partir con un conjunto de símbolos como el mencionado:

$$S = \{P, \ldots, f, \ldots, c, \ldots\},$$

En este conjunto tenemos listas finitas de símbolos para predicados, operaciones e individuos, respectivamente. La única exigencia adicional sobre estos símbolos es que cada símbolo para un predicado o para una operación tenga una *aridad* asociada; por ejemplo si P denota un predicado binario y f denota una operación de 3 argumentos (es decir una aridad 3), podemos escribirlos con sus argumentos explícitamente: $P(\cdot, \cdot)$, $f(\cdot, \cdot, \cdot)$.

Las proposiciones del lenguaje formal hablan sobre objetos, también formales que corresponden, en el contexto semántico, a individuos de los dominios de la interpretación, estos objetos son los *términos* del lenguaje.

Definición 3.28. *(S finito) Son* ***términos*** *del lenguaje de la FO con base en S, todas las sucesiones de símbolos construídas aplicando una cantidad finita de veces las siguientes reglas: 1. Toda variable es un término. 2. Todo nombre para individuo c, ($c \in S$) es término. 3. Si f es una operación n-aria ($f \in S$) y $t_1, t_2, \ldots t_n$ son términos, entonces $f(t_1, t_2, \ldots t_n)$*

En notación BNF podemos escribir:

$$t(S) ::= x \quad | \quad c \quad | \quad f(t_1, t_2, \ldots t_n)$$

la cual indica que un término t puede estar formado por elementos de S, donde x representa una variable, c una constante (o una función con aridad nula), y una operación f con una aridad $n > 0$.

Por ejemplo si $S = \{<, +, \cdot, 0, 1\}$, son términos, entre otros: x_2, 0, x_9, 1, $\cdot(x_1, +(x_2, 1))$. O bien, si n, f y g son símbolos de funciones de S, con aridad nula, unaria y binaria respectivamente; entonces $g(f(n), n)$ y $f(g(n, f(n)))$ son términos. Y si consideramos un conjunto de símbolos base a 0,1 con aridad nula, a s como unario, y a $+$, $-$, y a $*$ como binarios, entonces $*(-1, +(s(x), y)), x)$ es un término.

Los predicados atómicos pueden ser expresiones matemáticas suficientemente precisas cuyo tipo de datos sea booleano, por ejemplo: constantes booleanas, $\bot$, $\top$; una simple variable booleana b; expresiones aritméticas relacionales, $x + y = 7$, $x/y < z$; una función matemática conocida con valor booleano, $eof(f)$, $vacia(p)$

Los términos no son proposiciones potenciales, tan sólo denotan individuos de posibles dominios de interpretación, para expresar afirmaciones necesitamos las fórmulas del lenguaje, las cuales las definimos por inducción estructural como sigue:

$$\phi ::= P(t_1, t_2, \ldots t_n)|(\neg\phi)|(\phi \wedge \phi)|(\phi \vee \phi)|(\phi \rightarrow \phi)|(\forall x\phi|(\exists x\phi).$$

donde P , símbolo de un predicado de aridad $n \geq 1$, y t_i, términos, son elementos de S y x es una variable. Por ejemplo si $S_a = \{<, +, \cdot, 0, 1\}$ con las fórmulas del lenguaje formal podemos expresar propiedades de los números naturales:

$\forall x \forall y \forall z(x < y \wedge 0 < z \rightarrow x \cdot z < y \cdot z)$
$\forall x(x + 0 = x)$
$\forall x((0 < x \vee 0 = x) \wedge \exists y x < y)$

Para indicar los dominios asociados a los cuantificadores se indican como sigue: $(\forall \alpha \in D.P)$, donde D es el nombre del *dominio* de valores y P es un predicado. El siguiente ejemplo de predicado es sintácticamente correcto:

$$(\exists \phi \in \mathcal{N}.(\forall \beta \in \{1 \ldots n\}.\phi[\beta] \leq \phi))$$

Como se aprecia, en un predicado pueden aparecer variables del algoritmo que se esta especificando, en este caso se trata de variables *libres*; otras veces las variables van asociadas a un cuantificador universal, existencial o a otros de tipo matemático, tales como sumatorias o productos; tales variables intentan decir algo acerca de todos los valores de un cierto dominio, tales variables se denominan *mudas* o *ligadas*. Para distinguir las variables, tenemos la siguiente definición:

Definición 3.29. *Los conjuntos libres(P) y ligadas(P), correspondientes a los conjuntos de variables libres y ligadas de un predicado P, respectivamente, se definen a partir de la estructura sintáctica de P del modo siguiente:*

1. *libres(P) = variables(P); ligadas(P) = ∅, si P es atómico.*

2. $libres(\neg P) = libres(P)$; $ligadas(\neg P) = ligadas(P)$.
3. $libres(P \diamondsuit Q) = libres(P) \cup libres(Q)$.
4. $ligadas(P \diamondsuit Q) = ligadas(P) \cup ligadas(Q)$.
5. $libres(\partial x \in D.P) = libres(P) - x$.
6. $ligadas(\partial x \in D.P) = ligadas(P) \cup x$.

El símbolo $\diamondsuit$ denota cualquiera de los conectivos del conjunto $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ y el símbolo ∂ denota cualquiera de los cuantificadores $\forall$ y $\exists$. La D., antes de P, indica el dominio bajo el cual se debe considerar a P

por ejemplo si tenemos los predicados:

$P \equiv (x + y + z > 0)$, $Q \equiv (\exists y \in \mathcal{N}.P)$ y $R \equiv (\forall x \in \mathcal{N}.Q)$, es decir:
$R \equiv (\forall x \in \mathcal{N}.x + y + z > 0))$

aplicando la definición anterior, obtenemos los siguientes conjuntos de variables:

$libres(P) = \{x, y, z\}$; $ligadas(P) = \varnothing$.
$libres(Q) = \{x, z\}$; $ligadas(Q) = \{y\}$.
$libres(R) = \{z\}$; $ligadas(R) = \{x, y\}$.

La interpretación sintáctica de un predicado en FO utiliza la siguiente gramática:

Pred::=	Cuantif Pred | PredSimple
Cuantif ::=	$\{\forall \mid \exists\}$ Ident $\exists$ Ident
PredSimple ::=	ParteBicond | $\leftrightarrow$ ParteBicond |
ParteBicond ::=	ParteCond | $\rightarrow$ ParteCond |
ParteCond ::=	Termino | $\{\vee Termino\}^*$
Termino ::=	Factor | $\{\wedge Factor\}^*$
Factor ::=	$\neg$ Factor | PredAtomico | (Pred)

Por ejemplo tenemos la fórmula: $\forall x((P(x) \rightarrow Q(x)) \wedge R(x, y))$, las fórmulas de la lógica de predicados se pueden representar en forma de árboles de partición.

En cuanto a la semántica de la lógica FO, podemos comenzar por dar significado a los símbolos no lógicos del lenguaje. Sea $S = \{R, \ldots, f, \ldots, c, \ldots\}$ el conjunto de símbolos básicos. Estos símbolos se interpretan en estructuras.

Definición 3.30. *Dado $S = \{R, \ldots, f, \ldots, c, \ldots\}$ una **estructura** compatible con (o para) S es una tupla:*

$\mathfrak{U} = \langle A, R^A, \ldots, f^A, \ldots, c^A, \ldots \rangle$, *en la cual:*

- *A es un conjunto no vacio llamado **universo** o **dominio** de la estructura.*

- *Si R es símbolo para predicado n-ario, entonces R^A e la relación n-aria sobre A, es decir $R^A \subseteq A^n$ donde A^n es el producto cartesiano de A con si mismo, n veces.*
- *Si f es símbolo para operación n-aria, entoncesf^A es una operación (función) n-aria sobre A, es decir, $f^A : A^n \to A$.*
- *Si c es un símbolo para constante, entonces c^a es un elemento de A, un individuo particular del dominio: $c^A \in A$.*

Decimos que $R^A, \ldots, f^A, \ldots, c^A, \ldots$ son las interpretaciones de $R, \ldots, f, \ldots, c, \ldots$ en $\mathfrak{U}$.

Por ejemplo, si $S_{ar} = \{<, +, \cdot, 0, 1\}$ es el conjunto de símbolos para hablar sobre los números naturales, una interpretación natural para estos símbolos es provista por la estructura de los números naturales:

$$\mathfrak{N} = \langle \mathbb{N}, <^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}} \rangle.$$

los símbolos $<, +, \cdot, 0, 1$ del lenguaje objeto se interpretan como los objetos matemáticos que son, denotados en el metalenguaje (matemático usual) a través de $<^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}}$, respectivamente.

Notemos que:

- $<^{\mathbb{N}}$ es la relación binaria sobre $\mathbb{N}$, es decir, $<^{\mathbb{N}} \subseteq \mathbb{N} \times \mathbb{N}$.
- $+^{\mathbb{N}}, \cdot^{\mathbb{N}}$ son operaciones binarias sobre $\mathbb{N}$, es decir, $+^{\mathbb{N}}, \cdot^{\mathbb{N}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.
- $0^{\mathbb{N}}, 1^{\mathbb{N}} \in \mathbb{N}$, es decir, son elementos de $\mathbb{N}$

Un mismo lenguaje tiene muchas interpretaciones posibles, la lógica debe dar cuenta de esta diversidad. El concepto de verdad lógica, tiene que ver con una aseveración formalizada sea verdadera en cualquier contexto, en cualquier interpretación imaginable.

Ejemplo: Sea $S = \{P\}$, donde P es un predicado unario. Una estructura compatible con S es $\langle \mathbb{Z}, \mathbb{P} \rangle$, donde $\mathbb{P}$ es el conjunto de los números pares.

Para poder dar interpretación a los términos de $T(S)$; es decir, dado un conjunto de símbolos S, de la construcción del conjunto $T(S)$ de términos del lenguaje; partimos de una estructura:

$$\mathfrak{U} = \langle A, R^A, \ldots, f^A, \ldots, c^A, \ldots \rangle$$

que da significado a los símbolos de S, y damos significado (valores) a las variables a través de una *asignación*. Esta es una función del conjunto de variables al dominio de la estructura:

$$\beta : \{x_1, x_2, \ldots\} \to A,$$

es decir, β da valores, que están en el dominio de la estructura, a las variables del lenguaje.

Ejemplo: Consideremos nuevamente: $S = \{<, +, \cdot, 0, 1\}$, y la estructura:

$\mathfrak{N} = \langle \mathbb{N}, <^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}} \rangle$.

Una posible asignación es:

$$\beta_0 : \{x_1, x_2, \ldots\} \to \mathbb{N}$$

definida por: $\beta_0(x_i) := 2i \quad (\in \mathbb{N})$. Por ejemplo se tiene $\beta_0(x_7) := 14$ y de la misma manera para un término más complejo:

$$x_7 + ((1+1) \cdot x_3)$$

tenemos la siguiente interpretación:

$$14 +^{\mathbb{N}} ((1^{\mathbb{N}} + 1^{\mathbb{N}}) \cdot^{\mathbb{N}} 6) = 26.$$

Definición 3.31. *Una **interpretación** para $L(S)$ es una pareja $\mathcal{I} = (\mathfrak{U}, \beta)$, donde $\mathfrak{U}$ es una estructura compatible con S y β es una asignación sobre A. La interpretación de un término está basada en la estructura de términos:*

1. $\mathcal{I}(c) := c^A$
2. $\mathcal{I}(x) := \beta(x)$
3. $\mathcal{I}(f(t_1, \ldots t_n)) := f^A(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$

Ejemplo: Consideremos: $S_{ar} = \{<, +, \cdot, 0, 1\}$ y la interpretación:

$$\mathcal{I}_0 = (\mathfrak{N}, \beta_0); \quad \mathfrak{N} = \langle \mathbb{N}, <, +, \cdot, 0, 1 \rangle, \quad \beta_0 : x_i \mapsto 2i, \quad i = 1, 2, \ldots$$

Entonces para $x_7 + ((1+1) \cdot x_3)$ tenemos:

$$\begin{aligned} \mathcal{I}(x_7 + ((1+1) \cdot x_3)) &:= \beta_0(x_7) + (1+1) \cdot \beta_0(x_3) \\ &= 14 + 2 \cdot 6 \\ &= 26 \quad (\in \mathbb{N}). \end{aligned}$$

Siempre la interpretación de un término es un elemento del dominio de la estructura.

La siguiente noción es la de verdad de una fórmula en una interpretación. Consideremos un lenguaje de primer orden $L(S)$, necesitamos definir la relación $(I) \models \phi$, en la que $\mathcal{I} = (\mathfrak{U}, \beta)$ es una interptretación compatible con S y ϕ es una fórmula de $L(S)$, es decir la relación "$\mathcal{I}$ satisface, o hace verdadera a, o es modelo de, ϕ.

Definición 3.32. *$(I) \models \phi$ por inducción estructural en la fórmula ϕ de $L(S)$, para una interpretación $\mathcal{I} = (\mathfrak{U}, \beta)$ compatible con $L(S)$ arbitraria:*

1. *Si ϕ es una fórmula atómica de la forma $t = s$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad (I)(t) = (I)(s)$.
2. *Si ϕ es atómica de la forma $R(t_1, t_2, \ldots, t_n)$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad ((I)(t_1), \ldots, ((I)(t_n)) \in R^A$.

3. *Si ϕ es de la forma $\neg\psi$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad$ *no (es cierto que)*$(I) \models \psi$.

4. *Si ϕ es de la forma $(\psi \wedge \varphi)$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad (I) \models \psi$ *y* $(I) \models \varphi$.

5. *Si ϕ es de la forma $(\psi \vee \varphi)$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad (I) \models \psi$ *o* $(I) \models \varphi$.

6. *Si ϕ es de la forma $(\psi \rightarrow \varphi)$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad (I) \models \psi \Leftrightarrow (I) \models \varphi$.

7. *Si ϕ es de la forma $(\psi \leftrightarrow \varphi)$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad (I) \models \psi \Leftrightarrow (I) \models \varphi$.

8. *Si ϕ es de la forma $\exists x\psi$, entonces:*
   $(I) \models \phi \quad :\Leftrightarrow \quad$ *existe $a \in A$ tal que:* $(I), (a/x) \models \psi$, *donde:*

$$\beta(a/x)(y) = \begin{cases} \beta(y) & \text{si } y \neq x, \\ a & \text{si } y = x. \end{cases}$$

   Es decir, la nueva asignación coincide con β en los valores de todas las variables, excepto, posiblemente, en el valor de x, a la cual asigna el valor a.

9. *Si ϕ es de la forma $\forall x\psi$, entonces:*

$$\mathcal{I} \models \phi \quad := \quad \text{para todo } a \in A,\ (I), (a/x) \models \psi.$$

Por ejemplo consideremos $S = \{<, +, \cdot, 0, 1\}$ y la estructura $\mathfrak{R} = \langle \mathbb{R}, <, +, \cdot, 0, 1 \rangle$, y la asignación:

$$\beta : \{x_1, x_2, \ldots\} \rightarrow \mathbb{R}\text{y } ,x_i \mapsto \frac{i+1}{2}.$$

con éstas premisas formamos la interpretación $(I) = (\mathcal{R}, \beta)$, y si tenemos la fórmula:

$$\exists x_6(x_6 + x_8 < 0)$$

podemos examinar si es verdadera en la interpretación establecida:

$$\mathcal{I} \models \exists x_6(x_6 + x_8 < 0) \Leftrightarrow \text{exister} \in \mathbb{R}\text{tal que,:}$$

$$(\mathcal{R}, \beta(x_6/r)) \models x_6 + x_8 < 0.$$

Es decir, si y solo si existe $r \in \mathbb{R}$, tal, que: $r + \frac{8+1}{2} < 0$. Como esto es cierto, la fórmula original es verdadera en la interpretación dada.

3.2.3. Lógica de Segundo Orden o SO (Second Order logic)

La lógica de segundo orden generaliza la FO permitiendo la cuantificación sobre un nuevo tipo de variables, llamadas de segundo orden, que representan relaciones o funciones. Las variables de primer orden $x, y, z \ldots$ toman valores sobre un dominio D de una estructura, en tanto que las variables de segundo orden (de aridad n) son interpretadas por relaciones (de aridad n) sobre D o de funciones de D^n en D. Como en las secciones anteriores, los símbolos $R, S, T, \ldots$ se utilizan para las relaciones y $f, g, h, \ldots$ para las funciones. La aridad de estos objetos, cuando aparecen en una fórmula, esta dada por el contexto.

Sea L_0 un lenguaje de primer orden arbitrario y K la clase de todas las estructuras para este lenguaje. El lenguaje L se obtiene agregando a L_0 un conjunto $\{X, Y, Z, \ldots\}$ de variables de relaciones y un conjunto $\{f, g, h \ldots\}$ de variables de funciones. El conjunto de términos del lenguaje L esta construido como uno de primer orden utilizando los nuevos símbolos de funciones. El conjunto de las fórmulas atómicas esta definido como en FO utilizando los nuevos símbolos de las relaciones.

Definición 3.33. *El conjunto de las fórmulas de segundo orden es el conjunto más pequeño que contiene las fórmulas atómicas y se establece por los siguientes procedimientos de construcción:*

- *Los conectores ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow$) y los cuantificadores ($\exists, \forall$) sobre las variables de primer orden.*
- *Si X es una variable de relación y F es una fórmula, entonces $\exists X F$ y $\forall X F$ son fórmulas.*
- *Si f es una variable de función y F una fórmula, entonces $\exists f F$ y $\forall f X F$ son fórmulas.*

Por ejemplo, la fórmula siguiente es una fórmula de segundo orden en el lenguaje que incluye un símbolo de relación binario $\neg E$ y una variable de relación unaria S:

$$\exists S(\exists x S(x) \wedge \forall x \forall y(S(x) \wedge S(y) \wedge x \neq y \rightarrow Exy))$$

La definición de la satisfacilidad de una fórmula se generaliza agregando las condiciones que conciernen a la cuantificación de segundo orden:

Definición 3.34. *Sea U una estructura-L y F una fórmula de segundo orden.*

- *Si F es $\exists X \quad G$ en la que X es de aridad n, entonces $U \models \exists X G$, si existe una relación R^U de aridad n sobre U, tal que $(U, R^U) \models G$. Si F es $\forall X \quad G$, entonces $U \models \forall X \quad G$ si cualquiera que sea la relación R^U de aridad n sobre U, $(U, R^U) \models G$.*
- *Si F es $\exists f \quad G$ en la que f tiene m argumentos, entonces $U \models \exists f \quad G$, si existe una función f^U de U^m en U, tal que $(U, f^U) \models G$. Si F es $\forall f \quad G$, entonces $U \models \forall f \quad G$ si cualquiera que sea la función f^U de U^m en U, $(U, f^U) \models G$.*

Es importante notar que una fórmula que comienza por una negación, seguida por una serie de cuantificadores de segundo orden, es equivalente a una fórmula que comienza por una serie dual de cuantificadores de segundo orden, seguida por una negación, como en el caso de la lógica de primer orden, por ejemplo:

$$\neg R \forall S \exists T F(R, S, T) \leftrightarrow \forall R \exists S \forall T \neg F(R, S, T)$$

Ejemplo. Consideremos un lenguaje que contiene una relación binaria E y la clase de las gráficas.

1. La siguiente fórmula expresa la existencia de un subconjunto S no vacío, en el cual todas las parejas de vértices están conectadas por una arista de la gráfica. La restricción de la gráfica a este subconjunto constituye una subgráfica completa, llamada también una *clique*:
$$\exists S(\exists x S(x) \wedge \forall x \forall y((S(x) \wedge S(y) \wedge x \neq y) \rightarrow Exy))$$

2. La siguiente fórmula expresa que una gráfica es bipartita, es decir que el conjunto de sus vértices esta dividido en dos subconjuntos tales que para toda arista de la gráfica, si una de sus extremidades esta en uno de los subconjuntos, entonces la otra extremidad está en el otro subconjunto.
$$\exists S \exists T(\forall x((S(x) \vee T(x)) \wedge (S(x) \leftrightarrow \neg T(x))) \wedge \forall x \forall y(\exists xy \rightarrow (S(x) \leftrightarrow T(y))))$$

Definición 3.35. *La lógica SO Monádica (SOM) esta definida como la restricción de la SO que consiste en que todas las variables de segundo orden tienen aridad 1.*

Definición 3.36. *La lógica SO existencial (∃ SO) esta definida como la restricción de la SO que consiste en las formulas de la forma:*

$$\exists X_1 \ldots \exists X_n \quad \phi$$

donde ϕ no contiene ninguna cuantificación de segundo orden. Es decir, una fórmula inicia con un prefijo existencial de segundo orden $\exists X_1 \ldots \exists X_n$, y lo que le sigue es una fórmula ϕ de FO. Si, además, todas todas las X_i tienen aridad 1, a la restricción resultante se le denomina SO monádica existencial (∃ MSO).

Cabe mencionar que las lógicas ∃MSO y ∀MSO sus fórmulas son respectivamente de la forma:

$$\exists X_1 \ldots \exists X_n \quad \phi$$

y

$$\forall X_1 \ldots \forall X_n \quad \phi$$

en las que ϕ es de primer orden.

También se encuentran con la denominación Σ_1^1 monádico para la ∃MSO (Π_1^1 monádico para ∀ MSO), en general Σ_k^1 esta formado en fórmulas cuyo prefijo de cuantificadores de segundo

orden compuesta por k bloques, y el primer bloque corresponde al existencial. Por ejemplo, la fórmula:

$$\exists X_1 \exists X_2 \forall Y_1 \exists Z_1 \psi$$

es una fórmula Σ_3^1. Otro nombre bajo el que se denomina a $\exists MSO$ es el de NP monádico.

3.2.4. Lógica Temporal Lineal o LTL(Linear-time Temporal Logic)

La LTL es una lógica temporal con conectivos que permiten referirse al futuro, modela el tiempo como una secuencia lineal de estados, llamada trayectoria de cómputo, que se extiende de manera infinita con dirección al futuro. En general se consideran varias trayectorias, representando diferentes futuros posibles en la evolución del sistema.

Trabaja con un conjunto finito de fórmulas atómicas, tal como $\{p, q, r, \ldots\}$, que representan hechos atómicos que puede realizar un sistema en particular, por ejemplo: "Proceso de medición M3 liberado", "Contenido de contador C12 puesto a cero", etc.

Definición 3.37. *La sintaxis de una fórmula de la lógica LTL esta definida inductivamente, mediante notación BNF, como sigue:*

$$\begin{aligned}\phi ::= &\bot|\top|p|(\neg\phi)|(\phi \wedge \phi)|(\phi \vee \phi)|(\phi \rightarrow \phi)|(X\phi)|(F\phi)| \\ &(G\phi)|(\phi U \phi)|(\phi W \phi)|(\phi R \phi)\end{aligned}$$

En esta definición p corresponde a un elemento de un conjunto de proposiciones atómicas, el conectivo X significa "en el siguiente estado" (neXt state), el conectivo F significa "en algún estado en el futuro"(Future state), el conectivo G significa "'en todos los estados en el futuro" (Globally), en cuanto a los conectivos U, R y W son llamados "hasta que" (Until), "Liberado" (Release) y "hasta-que débil" (Weak-until) respectivamente. Algunas fórmulas bien formadas de esta lógica son:

$$(((Fp) \wedge (Gq)) \rightarrow (pWr))$$
$$(F(p \rightarrow (Gr)) \vee ((\neg q)Up))$$

En cuanto a la semántica de la LTL se puede señalar que su interpretación se presenta con base en un sistema de transición, como sigue:

Definición 3.38. *Un sistema de transición $\mathcal{M} = (S, \rightarrow, L)$ es un conjunto de estados S que esta dotado con una relación de transición $\rightarrow$ (relación binaria en S), tal que cada $s \in S$ tiene algún $s' \in S$ con $s \rightarrow s'$, y una función de etiquetado $L : S \rightarrow \mathcal{P}(Atoms)$.*

Los sistemas de transición, también llamados *modelos*, tienen un conjunto de estados S, una relación $\rightarrow$, que indica como el sistema puede ir de un estado a otro, y asociado con cada estado s se tiene el conjunto de proposiciones atómicas $L(s)$ con las cuales son verdaderas en ese estado particular. Por $\mathcal{P}(Atoms)$ se denota al conjunto potencia de $Atoms$, el cual consiste en un conjunto de descripciones atómicas. Por ejemplo, el conjunto de $Atoms = \{p, q\}$

corresponde a $\mathcal{P}(Atoms) = \{\varnothing, \{p\}, \{q\}, \{p, q\}\}$. Una forma de ver a L es que se trata de una asignación de valores de verdad para todos los átomos proposicionales. La diferencia con la valuación en la lógica de enunciados radica en que en la LTL se tiene que considerar *más de un estado*, por consiguiente la asignación depende del estado s en que se encuentre el sistema: $L(s)$ contiene todas los átomos que son verdaderos en el estado s.

Una representación para expresar toda la información acerca de un sistema $\mathcal{M}$ de transiciones finito, es una gráfica dirigida cuyos nodos corresponden a los estados, los cuales contienen todos los átomos de proposiciones que son verdaderos en cada uno de esos estados.

Definición 3.39. *Una trayectoria en un modelo $\mathcal{M} = (S, \rightarrow, L)$ es una secuencia infinita de estados $s_1, s_2, s_3, \ldots$ en S tal que, para toda $i \geq 1$, $s_i \rightarrow s_{i+1}$. Escribimos la trayectoria como $s_1 \rightarrow s_2 \rightarrow \ldots$. La notación para indicar que una trayectoria inicia en s_i es: π^i.*

Definición 3.40. *Sea $\mathcal{M} = (S, \rightarrow, L)$ un modelo y $\pi = s_1 \rightarrow \ldots$ sea una trayectoria en $\mathcal{M}$ Que π satisfaga una una formula LTL esta definido por la relación $\models$ como sigue:*

1. $\pi \models \top$
2. $\pi \not\models \bot$
3. $\pi \models p$ *si y sólo sí* $p \in L(s_1)$
4. $\pi \models \neg\phi$ *si y sólo sí* $\pi \not\models \phi$
5. $\pi \models \phi_1 \wedge \phi_2$ *si y sólo sí* $\pi \models \phi_1$ *y* $\pi \models \phi_2$
6. $\pi \models \phi_1 \vee \phi_2$ *si y sólo sí* $\pi \models \phi_1$ *o* $\pi \models \phi_2$
7. $\pi \models \phi_1 \rightarrow \phi_2$ *si y sólo sí* $\pi \models \phi_2$ *siempre que* $\pi \models \phi_1$
8. $\pi \models X\phi$ *si y sólo sí* $\pi^2 \models \phi$
9. $\pi \models G\phi$ *si y sólo sí para todo* ß ≥ 1, $\pi^i \models \phi$
10. $\pi \models F\phi$ *si y sólo sí para algún* $i \geq 1$ *tal que* $\pi^i \models \phi$.
11. $\pi \models \phi U \psi$ *si y sólo sí existe algún* $i \geq 1$ *tal que* $\pi^i \models \psi$ *y para toda* $j = 1, \ldots, i-1$ *tenemos* $\pi^j \models \phi$
12. $\pi \models \phi W \psi$ *si y sólo sí ya sea existe algún* $i \geq 1$ *tal que* $\pi^i \models \psi$ *y para toda* $j = 1, \ldots, i-1$ *tenemos* $\pi^j \models \phi$, *o para toda* $k \geq 1$ *tenemos que* $\pi^k \models \phi$
13. $\pi \models \phi R \psi$ *si y sólo sí ya sea que existe algún* $i \geq 1$ *tal que* $\pi^i \models \phi$ *y para toda* $j = 1, \ldots, i$ *tenemos* $\pi^j \models \psi$, *o para toda* $k \geq 1$ *tenemos que* $\pi^k \models \psi$.

Algunos ejemplos pueden ser:

$$G((\neg pasaporte \vee \neg tiket) \quad \rightarrow \quad X \neg abordar\_vuelo)$$

$$((x = 0) \wedge add3) \quad \rightarrow \quad X(x = 3)$$
$$solicitud \quad \rightarrow \quad repetir \quad U \quad acuce\_de\_recibo$$

Definición 3.41. *Decimos que dos fórmulas LTL ϕ y ψ son semánticamente equivalentes, o simplemente equivalentes, lo que se denota como $\phi \equiv \psi$, si para todos los modelos $\mathcal{M}$ y todas las trayectorias ϕ en $\mathcal{M} : \pi \models \phi$ si y sólo sí $\pi \models \psi$.*

Algunas de las equivalencias relevantes en LTL son:

$$\begin{aligned}
\neg G\phi &\equiv F\neg\phi \\
\neg F\phi &\equiv G\neg\phi \\
\neg X\phi &\equiv X\neg\phi \\
\neg(\phi U\psi) &\equiv \neg\phi R\neg\psi \\
\neg(\phi R\psi) &\equiv \neg\phi U\neg\psi \\
F(\phi \vee \psi) &\equiv F\phi \vee F\psi \\
G(\phi \wedge \psi) &\equiv G\phi \wedge G\psi \\
F\phi &\equiv \top U\phi \\
G\phi &\equiv \bot R\phi \\
\phi U\psi &\equiv \phi W\psi \wedge F\psi \\
\phi W\psi &\equiv \phi U\psi \vee G\phi \\
\phi W\psi &\equiv \psi R(\phi \vee \psi) \\
\phi R\psi &\equiv \psi W(\phi \vee \psi).
\end{aligned}$$

3.2.5. Lógica Computacional Ramificada o CTL (Computational Tree Logic)

La lógica CTL es una lógica modal con ramificación en el tiempo, esto representa que su modelo de tiempo es arborescente en la cual el futuro no esta determinada; se tienen diferentes trayectorias en el futuro, y cualquiera de ellas puede corresponder a la trayectoria "real" que es ejecutada.

Como con en el análisis de las otras lógicas, definiremos su sintaxis a partir de un conjunto fijo de fórmulas (o descripciones), tales como $p, q, r, \ldots$, o $p_1, p_2, \ldots$.

Definición 3.42. *La sintaxis de una fórmula de la lógica CTL esta definida inductivamente, mediante notación BNF, como sigue:*

$$\begin{aligned}
\phi ::= &\bot|\top|p|(\neg\phi)|(\phi \wedge \phi))|(\phi \vee \phi)|(\phi \rightarrow \phi)|(AX\phi)|(EX\phi)| \\
&(AF\phi)|(EF\phi)|(AG\phi)|(EG\phi)|(A[\phi U\phi])|(E[\phi U\phi])
\end{aligned}$$

Donde p corresponde a un elemento de un conjunto de fórmulas atómicas, el conectivo A significa "en todas las trayectorias" (*necesariamente*), el conectivo E significa "al menos en una trayectoria"(*posiblemente*). El segundo conjunto de cuantificadores temporales, X, F, G y U, representan "en el siguiente estado" (neXt state), "en algún estado en el futuro " (Future state), "en todos los estados en el futuro" (Globally) y "hasta que" (Until); respectivamente. El par de símbolos en $A[\phi U \phi]$ y en $E[\phi U \phi]$, corresponde respectivamente a los conectivos binarios AU y a EU, y deben manejarse de manera indivisible. Los símbolos X, F, G, y U no pueden aparecer sin estar precedidas por una A o por una E, de la misma manera, toda A o E deben estar acompañadas por alguno de los símbolos X, F, G, o U. Generalmente W "hasta-que débil" (Weak-until) y R "liberar" (Release) no se incluyen en CTL pero en caso necesario se pueden derivar.

Las prioridades vinculadas para los conectivos CTL si no se indica algo en contrario rigen como en las lógicas proposicional o de predicados: los conectivos unarios ($\neg$,AG, EG, AF, EF, AX, y EX), reune el más cercano. El próximo, en el orden que aparecen, $\wedge$ y $\vee$; y después los siguientes $\rightarrow$, AU y EU. Por ejemplo de acuerdo a esta convención la siguiente es una fbf:

$$AG(q \rightarrow EGr)$$

no se puede escribir como:

$$AGq \rightarrow EGr \quad \text{ya que ésta última representa a: } (AGq) \rightarrow (EGr)$$

Un ejemplo de una expresión que no corresponde a una fbf en CTL es la siguiente:

$$A[(rUq) \wedge (pUr)],$$

esto se debe a que la sintaxis no nos permite colocar directamente una conectiva booleana, en este caso $\wedge$, en el interior de $A[\ \ ]$ o de $E[\ \ ]$. A la aparición de una A o de una E le debe seguir X, F, G, o U; y cuando son seguidas por U debe tener la forma $A[(\phi U \phi]$.

Notemos que los conectivos binarios AU y EU se pueden escribir con notaciones puras infija o prefija como sigue:

$$\phi_1 AU \phi_2, \quad \text{o} \quad AU(\phi_1, \phi_2)$$

Definición 3.43. *Una sub-fórmula derivada de una fórmula ϕ de CTL es una fórmula ψ cuyo árbol de separación sintáctica (parse tree) corresponde a un sub-árbol del árbol de separación sintáctica de ϕ.*

Por ejemplo el árbol de separación sintáctica de la fórmula CTL:

$$A[AX\neg pU \quad E[EX(p \wedge q)U\neg p]]$$

se puede representar en forma gráfica para visualizar este proceso de análisis.

En cuanto a la semántica de la CTL se puede señalar que su interpretación se presenta también sobre un sistema de transición.

Sea $\mathcal{M} = (S, \rightarrow, L)$ tal modelo, $s \in S$ y ϕ una formula CTL. La definición de ya sea $\mathcal{M}, s \models \phi$ se cumpla es recursiva en la estructura de ϕ, y puede ser entendida como sigue:

Definición 3.44. *Sea $\mathcal{M} = (S, \to, L)$ un modelo para CTL, s en S, ϕ una fórmula CTL. La relación $\mathcal{M}, s \models \phi$ esta definida por inducción estructural sobre ϕ:*

1. $\mathcal{M}, s \models \top$, *y* $\mathcal{M}, s \not\models \bot$
2. $\mathcal{M}, s \models p$ *si y sólo sí* $p \in L(s)$
3. $\mathcal{M}, s \models \neg\phi$ *si y sólo sí* $\mathcal{M}, s \not\models \phi$
4. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ *si y sólo sí* $\mathcal{M}, s \models \phi_1$ *y* $\mathcal{M}, s \models \phi_2$
5. $\mathcal{M}, s \models \phi_1 \vee \phi_2$ *si y sólo sí* $\mathcal{M}, s \models \phi_1$ *o* $\mathcal{M}, s \models \phi_2$
6. $\mathcal{M}, s \models \phi_1 \to \phi_2$ *si y sólo sí* $\mathcal{M}, s \not\models \phi_1$ *o* $\mathcal{M}, s \models \phi_2$
7. $\mathcal{M}, s \models AX\phi$ *si y sólo sí para todo* s_1 *tal que* $s \to s_1$ *tenemos* $\mathcal{M}, s_1 \models \phi$. *Por consiguiente, AX nos indica "en todos los siguientes estados".*
8. $\mathcal{M}, s \models EX\phi$ *si y sólo sí para algún* s_1 *tal que* $s \to s_1$ *tenemos* $\mathcal{M}, s_1 \models \phi$. *Por consiguiente, EX nos indica "en alguno de los siguientes estados". E es dual de A exactamente de la manera que* $\exists$ *lo es para* $\forall$ *en FO.*
9. $\mathcal{M}, s \models AG\phi$ *se cumple si y sólo sí para todas las trayectorias* $s_1 \to s_2 \to s_3 \to, \ldots$, *donde* s_1 *es igual a* s, *y toda* s_i *a lo largo de la trayectoria, tenemos* $\mathcal{M}, s_i \models \phi$. *Notemos que en cualquier trayectoria se incluye también el estado inicial* s.
10. $\mathcal{M}, s \models EG\phi$ *se cumple si y sólo sí existe al menos una trayectoria* $s_1 \to s_2 \to s_3 \to, \ldots$, *donde* s_1 *es igual a* s, *y toda* s_i *a lo largo de la trayectoria, tenemos* $\mathcal{M}, s_i \models \phi$.
11. $\mathcal{M}, s \models AF$ *se cumple* ϕ *se cumple ssi para todas las trayectorias* $s_1 \to s_2 \to s_3 \to, \ldots$, *donde* s_1 *es igual a* s, *existe alguna* s_i *tal que* $\mathcal{M}, s_i \models \phi$.
12. $\mathcal{M}, s \models EF\phi$ *se cumple si y sólo sí existe una trayectoria* $s_1 \to s_2 \to s_3 \to, \ldots$, *donde* s_1 *es igual a* s, *y para alguna* s_i *a lo largo de la trayectoria tenemos que* $\mathcal{M}, s_i \models \phi$.
13. $\mathcal{M}, s \models A[\phi_1 U \phi_2]$ *se cumple si y sólo sí para todas las trayectorias* $s_1 \to s_2 \to s_3 \to, \ldots$, *donde* s_1 *es igual a* s, *esta trayectoria satisface* $\phi_1 U \phi_2$, *es decir existe alguna* s_i *a lo largo de la trayectoria, tal que* $\mathcal{M}, s_i \models \phi_2$, *y para toda* $j < i$, *tenemos que* $\mathcal{M}, s_j \models \phi_1$.
14. $\mathcal{M}, s \models E[\phi_1 U \phi_2]$ *se cumple si y sólo sí si existe alguna trayectoria* $s_1 \to s_2 \to s_3 \to, \ldots$, *donde* s_1 *es igual a* s, *y esta trayectoria satisface* $\phi_1 U \phi_2$, *es decir existe alguna* s_i *a lo largo de la trayectoria, tal que* $\mathcal{M}, s_i \models \phi_2$, *y para toda* $j < i$, *tenemos que* $\mathcal{M}, s_j \models \phi_1$ $\phi_1 U \phi_2$, *es decir existe alguna* s_i *a lo largo de la trayectoria, tal que* $\mathcal{M}, s_j \models \phi_1$.

Dos fórmulas CTL ϕ y ψ se dice que son semánticamente equivalentes si cualquier estado en cualquier modelo que satisface una de ellas también satisface a la otra; esto se denota por $\phi \equiv \psi$.

Como ejemplo se pueden mencionar las siguientes equivalencias:

$$\begin{aligned}\neg AF\phi &\equiv EG\neg\phi\\ \neg EF\phi &\equiv AG\neg\phi\\ \neg AX\phi &\equiv EX\neg\phi\\ AF\phi &\equiv A[\top U\phi]\\ EF\phi &\equiv E[\top U\phi]\end{aligned}$$

3.3. Automatas de Árbol

Un árbol es una gráfica no dirigida sin ciclos formada por un conjunto de vértices llamados *nodos*, unidos entre sí por aristas *padre-hijo*. Un nodo tiene como máximo un padre y cero o más hijos, las aristas conectan a los padres con cada uno de sus hijos. Tiene un sólo nodo, la *raíz*, que no tiene padre, los nodos sin hijos se llaman *hojas*, y los nodos que no son hojas se denominan *nodos interiores*.

El hijo de un nodo es un *descendiente* de ese nodo, el padre de un nodo es un *ascendiente* o *antepasado*, todo nodo puede considerarse ascendente y descendiente de sí mismo. Los hijos de un nodo se denominan hermanos y estos se ordenan de izquierda a derecha, si el nodo N esta a la izquierda del nodo M, entonces se considera que todos los descendientes de N están a la izquierda del nodo M. El grado o rango de un árbol es el número máximo de hijos que tienen sus subarboles, por ejemplo el grado de un árbol binario es 2.

Una vez que se ubica el nodo raíz, el árbol se puede recorrer como una estructura recursiva, cada uno de los nodos u en el árbol constituyen en si mismos la raíz de un *subárbol* $T(u)$. El conjunto de todos los nodos v cuya única trayectoria desde v a la raíz van hacia u. El árbol se puede recorrer de abajo hacia arriba al empezar por las hojas (subarbol con un nodo) e ir a subarboles cada vez mayores, hasta cubrir todo el árbol (subarbol de la raíz).

Con la finalidad de formalizar los conceptos acerca de los autómatas que tienen la estructura de árbol, a continuación establecemos los términos y notaciones propuestos en [CD+07]. Como en las secciones anteriores, con $\mathbf{N}$ representamos el conjunto de los enteros positivos, y al conjunto de cadenas finitas sobre $\mathbf{N}$ por $\mathbf{N}^*$, y la cadena vacia por ϵ

Un ***alfabeto con rango*** es una pareja $(\mathcal{F}, Aridad)$ donde $\mathcal{F}$ es un conjunto finito y $Aridad$ es un mapeo $\mathcal{F} \to \mathbf{N}$. La ***aridad*** de un símbolo $f \in \mathcal{F}$ es $Aridad(f)$. El conjunto de simbolos de aridad p se denota por $\mathcal{F}_p$. Los elementos de aridad $0, 1, \ldots, p$ son respectivamente llamados simbolos constantes, unarios, binarios,...,p-arios. Asumimos que $\mathcal{F}$ contiene al menos una constante, por ejemplo $f(,)$ corresponde a una declaración para un símolo binario f.

Sea $\mathcal{X}$ un conjunto de constantes llamadas ***variables***, asumimos que los conjuntos $\mathcal{X}$ y $\mathcal{F}_0$ son disjuntos. El conjunto $T(\mathcal{F},\mathcal{X})$ de ***terminos*** sobre un alfabeto con rango $\mathcal{F}$ y el conjunto de variables $\mathcal{X}$ es el conjunto más pequeño definido por:

- $\mathcal{F}_0 \subseteq T(\mathcal{F},\mathcal{X})$ y

- $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$ y
- si $p \geq 1$, $f \in \mathcal{F}_p$ y $t_1, \ldots, t_p \in T(\mathcal{F},\mathcal{X})$, entonces $f(t_1, \ldots, t_p) \in T(\mathcal{F}, \mathcal{X})$.

Si $\mathcal{X} = \varnothing$ entonces $T(\mathcal{F}, \mathcal{X})$ también se puede escibir como $T(\mathcal{F})$ los terminos en $T(\mathcal{F})$ se denominan ***terminos base*** (ground terms). Un termino t en $T(\mathcal{F}, \mathcal{X})$ es ***lineal*** si cada una de las variables aparece en t una sola vez cuando mucho.

Un ***árbol*** ordenado finito sobre un conjunto de etiquetas E es un mapeo desde un conjunto cerrado de prefijos $Pos(t) \subseteq \mathbf{N}^*$ hacia E. Por lo tanto un término $t \in T(\mathcal{F}, \mathcal{X})$ puede ser considerado como un árbol con rango ordenado, cuyas hojas están etiquetadas con símbolos variables o constantes y los nodos internos están etiquetados con simbolos de aridad positiva, con grado exterior igual a la aridad de la etiqueta, es decir, un término $t \in T(\mathcal{F}, \mathcal{X})$ puede también definirse como una función parcial $t : \mathbf{N}^* \to \mathcal{F} \cup \mathcal{X}$ con dominio $Pos(t)$ que satisface las siguientes propiedades:

- $Pos(t)$ no esta vacio y cerrado en prefijo.
- $\forall p \in Pos(t)$, si $t(p) \in \mathcal{F}_n$, $n \geq 1$, entonces $\{j | pj \in Pos(t)\} = \{1, 2, \ldots, n\}$.
- $\forall p \in Pos(t)$, si $t(p) \in \mathcal{X} \cup \mathcal{F}_0$, entonces $\{j | pj \in Pos(t)\} = \emptyset$.

Cada uno de los elementos en $Pos(t)$ se denomina una ***posición***. Una ***posición de frontera*** es una posición p tal que $\forall j \in N$, $pj \notin Pos(t)$. El conjunto de posiciones de frontera se denota por $\mathcal{F}Pos(t)$. Toda posición p en t tal que $t(p) \in \mathcal{X}$ se llama una ***posición de variable***. El conjunto de posiciones de variable de p se nota por $\mathcal{V}Pos(t)$. Por $Head(t)$ representamos al ***símbolo raíz*** de t el cual se define como $Head(t) = t(\epsilon)$.

Definición 3.45. ***Un Autómata de Árbol Finito No Determinista*** *(NFTA) sobre $\mathcal{F}$ es una tupla $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ donde Q es un conjunto finito de estados (unarios), $Q_f \subseteq Q$ es un conjunto de estados finales, y Δ es un conjunto de reglas de transición del tipo siguiente:*

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)),$$

donde $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in \mathcal{X}$

El autómata en $\mathcal{F}$ evoluciona sobre términos base de $\mathcal{F}$. Un automata empieza en las hojas y se mueve hacia arriba, asociando inductivamente a lo largo del proceso un estado con cada subtérmino. Notemos que en un NFTA no se tiene un estado inicial, pero, cuando $n = 0$, es decir cuando el símbolo es una constante a, una regla de transición tiene la forma $a \to q(a)$. Por consiguiente, las reglas de transición para los simbolos constantes pueden ser consideradas como las "reglas iniciales". Si los subterminos directos $u_1, u_2, \ldots, u_n$ de $t = f(u_1, u_2, \ldots, u_n)$ estan etiquetados con los estados $q_1, q_2, \ldots, q_n$, entonces el término t estará etiquetado por algún estado q con $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \to q(f(x_1, x_2, \ldots, x_n)) \in \Delta$. Por último definimos formalmente la relación de movimiento para un NFTA como sigue: Sea $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$

para un NFTA sobre $\mathcal{F}$. La ***relación de movimiento*** (move relation) $\rightarrow_{\mathcal{A}}$ se define como: sea $t, t' \in T(\mathcal{F} \cup Q)$,

$$t \rightarrow_{\mathcal{A}} t' \Leftrightarrow \begin{cases} \exists C \in \mathcal{C}(\mathcal{F} \cup Q), \exists u_1, u_2, \ldots, u_n \in T(\mathcal{F}), \\ \exists f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow q(f(x_1, \ldots, x_n)) \in \Delta, \\ t = C[f(q_1(u_1), q_2(u_2), \ldots, q_n(u_n))], \\ t' = C[q(f(u_1, u_2, \ldots, u_n))]. \end{cases}$$

$\rightarrow^*_{\mathcal{A}}$ se refiere a una cerradura reflexiva y transitiva de $\rightarrow_{\mathcal{A}}$.

Definición 3.46. *Un automata de árbol $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ es determinista si no se tienen dos reglas con el mismo lado izquierdo ni reglas-ϵ.*

Un DFTA no es ambiguo, y tiene a lo más una regla para cada término base, es decir para todo termino base t, se tiene cuando mucho un estado q tal que $t \rightarrow^*_{\mathcal{A}} q$.

Un NFTA se dice que es ***completo*** si existe al menos una regla $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow q \in \Delta$ para toda $n \geq 0$, $f \in \mathcal{F}_n$, y $q_1, q_2, \ldots, q_n \in Q$. Notemos que para un DFTA se tiene exactamente una corrida para cada término base.

3.4. Relaciones entre Clases de Complejidad

3.4.1. Las clases P y NP

Las clases P y NP establecen una cota polinomial sobre el tiempo y entonces nos refieren a máquinas acotadas en tiempo polinomial; de la misma manera las clases L y NL definen máquinas acotadas en espacio logaritmico, mientras que PSPACE establece máquinas acotadas en espacio polinomial.

Dentro de estas dos clases la más importante es la P que incluye a los problemas decidibles en tiempo polinomial sobre una máquina de Turing determinista. La clase NP, corresponde a la clase de problemas para el modelo de máquina no determinista, permite tomar en cuenta numerosos problemas de cálculo para los cuales no se conoce ningún algoritmo en tiempo polinomial; la cuestión de saber si estas dos clases son efectivamente distintas ($\mathsf{P} \neq \mathsf{NP}$) es una de las conjeturas mayores de la teoría de la complejidad.

La clase de complejidad polinomial P incluye los algoritmos que resuelven los problemas en una cantidad razonable de tiempo, ya que tienen un orden de complejidad que puede expresarse mediante una ecuación polinomial. En algunos casos, tenemos algoritmos que tienen tiempo lineal, como la búsqueda secuencial, donde si se duplica la lista por ordenar, la cantidad de tiempo que el algoritmo requiere también se duplica; tenemos, por otra parte, algoritmos que tienen tiempo $\mathcal{O}(n^2)$, como en los algoritmos de ordenación, donde si el tamaño de la lista se duplica, la cantidad de tiempo del algoritmo se incrementa por un factor de 4. Y podemos encontrar algoritmos que tienen tiempo $\mathcal{O}(n^3)$, como en el caso de multiplicación de matrices, en las que el tamaño de la matriz se duplica , la cantidad de tiempo del algoritmo tomará un incremento por un factor de 8. Aunque estos incrementos pueden ser significativos, permanecen

relativamente bajo control. De hecho, todos estos algoritmos tienen una complejidad dentro de $\mathcal{O}(n^3)$, recordemos que al mencionar una función $g(n)$ que esta en $\mathcal{O}(f(n))$ significa que $g(n)$ no crece más rápido que $f(n)$, entonces, n^2 esta en $\mathcal{O}(n^3)$. La cota inferior, sin embargo, consiste en obtener una solución exacta para aquellos problemas dentro de una cantidad razonable de tiempo, los problemas para los cuales se tiene un algoritmo ejecutable en tiempo polinomial son llamados tratables.

Sin embargo encontramos un conjunto de problemas que tienen que ejecutarse en tiempos que son del orden factorial $\mathcal{O}(n!)$ o exponencial $\mathcal{O}(x^n)(x \geq 2)$. En otras palabras, se trata de problemas para los cuales no hay algoritmo conocido que los resuelva en una cantidad de tiempo razonable de tiempo, estos problemas están en la clase NP, la cual corresponde a una complejidad con tiempo no determinista polinomial, se dice entonces que esta clase incluye problemas que no son tratables. A pesar de que estos problemas toman tiempos fuera de las espectativas de la tratabilidad, no podemos dejarlos a un lado debido a que nos los encuentramos en aplicaciones importantes, por ejemplo, para decidir una ruta eficiente en transportes de reparto, para desarrollar una programación de exámenes razonable, o para programar tareas cuyas fechas de cumplimento sean eficientemente distribuidas.

Al revisar el conjunto clásico de problemas en la clase NP, el aspecto por hacer notar es que solamente los algoritmos deterministas, que se conocen, resuelven los problemas con una complejidad de orden exponencial o factorial, es decir algunos de estos problemas tienen una complejidad dada por 2^n, donde n es el número de valores de entrada; en este caso, cada vez que agregamos un valor de entrada adicional, la cantidad de tiempo que el algoritmo necesita para resolver el problema será doble: si ejecuta 1,024 operaciones para resolver el problema con una entrada de 10 elementos, requerirá 2,048 operaciones para resolver el problema con una entrada de 11 elementos. Esto representa un incremento muy significativo en tiempo correspondiente a un pequeño incremento en el tamaño de la entrada.

El nombre de *no determinista polinómico* para problemas de la clase NP proviene de un proceso algorítmico de dos etapas para resolverlos. En la primera etapa se tiene un proceso no determinista que genera una solución posible para el problema, puede considerarse como una propuesta aleatoria, que en algunos casos será atinada (con la solución óptima o cerca de ésta) y en otras ocasiones será incorrecta (una lejana de la solución óptima). La segunda etapa opera a la salida de la primera y determina si es una solución verdadera. Individualmente ambas etapas trabajan en tiempo polinómico, el aspecto no determinista radica en que no conocemos cuantas veces se necesita repetir el proceso antes de obtener una solución aceptable. Es decir, no obstante que las etapas individualmente sean de orden polinómico, necesitamos llamarlas un número de veces exponencial o factorial, lo cual nos lleva, en el caso asíntótico a tiempos de tamaño no tratables.

Uno de los problemas clásicos en la clase NP es el problema del agente viajero o TSP (Traveling SalesPerson). En este problema, tenemos un conjunto finito de ciudades y un "costo" al viajar entre cada par de ciudades por visitar. El objetivo consiste en determinar el orden en que debemos recorrer todas y cada una de las ciudades en una sola ocasión, regresando a y terminando en la misma ciudad de partida, obteniendo simultáneamente el costo total mínimo del viaje. Este problema se puede aplicar a una serie de casos concretos y prácticos, como determinar el

orden que deben recorrerse las calles para realizar la recolección eficiente de basura, o decidir el itinerario que un transporte debe recorrer para realizar las entregas en el tiempo más corto posible, o seleccionar la ruta que debe tomar un paquete de datos para que su información sea transmitida a todos los nodos de una red de la manera más expedita. Cuando tenemos 8 ciudades, se tienen 40,320 combinaciones de ciudades, y cuando el numero de ciudades por visitar es de 10, tendremos 3,628,800 combinaciones posibles de recorrido. Para encontrar la ruta más eficiente, tenemos que examinar todas estas posibilidades. Supongamos que contamos con un algoritmo que puede calcular el costo de viajar entre un conjunto de 15 ciudades en un orden determinado. Si este algoritmo puede efectuar 100 de estos cálculos por segundo, tomará alrededor de *cuatro siglos* examinar todas las combinaciones de estas 15 ciudades y encontrar la de mejor costo total o la más rápida. Aún si contamos con 400 computadoras procesando el problema, tomaría alrededor de un año. Si requerimos incrementar a 20 ciudades, necesitaremos *mil millones de computadoras* trabajando en paralelo por un lapso de 9 meses para encontrar la ruta más eficiente. En este contexto, resulta más económico, en términos prácticos, viajar por todas las ciudades en cualquier orden determinado intuitivamente, que esperar a que el algoritmo encuentre el mejor recorrido.

Si revisamos el problema del agente viajero bajo la similitud de éste con los algoritmos de gráficas: Cada ciudad puede representarse como un nodo de la gráfica, la posibilidad de viajar entre dos ciudades se puede representar mediante las aristas de la gráfica, y el costo de viajar entre ellas se puede representar mediante un peso asignado a cada arista. Esto nos puede llevar a pensar que el algoritmo que calcula la ruta más corta podría resolver el problema, pero no es así. ¿Cuáles son los dos requerimientos del problema del agente viajero que lo hace diferente al problema de la ruta más corta? El primero consiste en que se deben visitar todos los nodos y el algoritmo de la ruta más corta nos indica la manera más rápida de hacer el recorrido entre dos nodos. Si tratamos de utilizar en forma de piezas múltiples el producto del algoritmo de la ruta más corta, puede encontrar que, comparando las trayectorias obtenidas, un nodo se visita más de una vez. La segunda diferencia radica en que esperamos regresar al punto de inicio en el problema del agente viajero, mientras que en el algoritmo de la trayectoria más corta no se tiene esta condición.

¿Es posible construir la trayectoria más corta sin buscar sobre todas las trayectorias posibles?. En este punto, nadie ha sido capaz de implementar la construcción de un algoritmo que efectivamente no requiera verificar todas las trayectorias potenciales. La situación en la que el número de ciudades es pequeño se puede resolver rápidamente, pero una instancia de este problema (con una entrada restringida) que pueda ejecutarse rápidamente no significa que exista tal algoritmo, capaz de realizar eficientemente todas las instancias del problema. El análisis de la gran cantidad de posibles combinaciones de los nodos nos muestra que un algoritmo determinista apto, para examinar todas las posibilidades, tomará inevitablemente un tiempo extremadamente grande para computarlo.

La demostración que un algoritmo esta en la clase NP, necesita establecer de que manera se puede implantar el proceso de solución a dos etapas que siguan las premisas ya descritas. Para el problema del agente viajero, por ejemplo, la primera etapa deberá generar de manera no determinista una lista de estas ciudades, debido a que el proceso es no determinista cada vez que se ejecuta genera una ordenación diferente de ciudades. Este proceso de producción

(o adivinatorio) se puede realizar en tiempo polinomial, porque podemos mantener una lista con los nombres de las ciudades, generar un número aleatorio y sacar el nombre de la ciudad correspondiente, y a continuación eliminar el nombre de la lista para evitar que aparezca dos veces; este proceso se ejecuta en el orden de $\mathcal{O}(n)$ siendo n la cantidad de ciudades. La segunda etapa estará encargada de determinar el costo de viajar por las ciudades, en el orden propuesto por la primera etapa, se determina simplemente buscando el costo para cada par sucesivo de ciudades dentro de la lista, lo cual representa una complejidad en el peor de los casos de $\mathcal{O}(n^2)$. Debido a que ambas etapas son de complejidad de orden polinomial, podemos asegurar que el problema del agente viajero esta en la clase NP. Notemos que la cantidad potencial de veces que este proceso de solución debe ejecutarse es la causa que se consuma tal cantidad de tiempo.

En este punto, podemos notar que se tiene la posibilidad de aplicar este proceso de dos etapas a cualquier algoritmo, incluso para los de la clase P. Por ejemplo, podemos realizar la ordenación de una lista: proponemos una lista aleatoria a partir de la original y a continuación verificamos si los elementos de la lista están colocados en orden decreciente. ¿Esto no hace que el problema de la ordenación sea miembro de la clase NP? Si en efecto. La diferencia entre un problema de la clase P y uno de la clase NP es que, en el primer caso también tenemos un algoritmo que corre en tiempo polinomial, mientras que el segundo caso no se ha encontrado. Estas observaciones nos llevan a concluir que:

$$\mathsf{P} \subseteq \mathsf{NP}$$

La pregunta natural y fundamental es si la inclusión inversa es verdadera:

$$¿\mathsf{NP} \subseteq \mathsf{P}?$$

Si la respuesta fuese positiva nos llevaría a demostrar que P=NP, lo que implicaria que todos los problemas identificados dentro de la clase NP tienen una solución determinista en tiempo polinomial. Como para ninguno de ellos se ha encontado todavia, ni se ha podido probar que exista (aunque no se exhiba), se considera muy poco probable que exista, y en consecuencia se presume que la inclusión inversa no es verdadera:

Conjetura de Cook: $\mathsf{P} \subsetneq \mathsf{NP}$, es decir, $\mathsf{P} \neq \mathsf{NP}$.

Este es el problema abierto más importante y famoso en la ciencia de la computación.

El término de NP-completo se utiliza para describir el grado de dificultad de los problemas dentro de la clase NP, estos problemas se señalan en particular debido a que si en cualquier momento, se conjetura, somos capaces de encontrar un algoritmo determinista en tiempo polinomial para resolver uno de ellos, implicará que todos los problemas en NP deben tener algoritmos deterministas en tiempo polinomial.

Definición 3.47. *Un problema de decisión L es NP-completo si:*

1. $L \in \mathsf{NP}$ *y*

2. *Todo problema de la clase NP es reducible en tiempo polinomial a L: para todo* $L' \in$ *NP, se tiene:*

$$L' \leq_P L$$

Recordemos que la clase NP contiene problemas que toman un tiempo extremadamente grande para obtener una respuesta, ya que no se ha encontrado un algoritmo capaz de resolverlos en tiempos más razonables. Si reexaminamos el problema del agente viajero de manera diferente, quizás estemos en posibilidad de desarrollar un algoritmo determinista que pueda resolverlo en tiempo polinomial. Esto mismo se podría pensar acerca de cualquiera de los problemas que se ha demostrado están en esta clase.

Estamos seguros que un problema es NP-completo al demostrar que todos los demás problemas en la clase NP se pueden transformar en el, esto no representa una tarea tan abrumadora como parece, debido a que no tenemos que hacer esto para todo problema NP. En lugar de esto, si tenemos un problema NP, que denominaremos A, podemos mostrar que esta en NP-completo al reducir un problema NP-completo, que llamaremos B, al problema A. Debido a que B es NP-completo, todo problema en NP se puede transformar al problema B. De tal manera, que al reducir B a A, efectivamente demostramos que todos los problemas NP se pueden transformar en A mediante un proceso a dos etapas que primero lo transforman a B. Por lo tanto, hemos constatado que nuestro problema NP A es NP-completo.

Si hemos podido realizar una reducción a un algoritmo en tiempo polinomial; ahora planteamos hacer una reducción para un problema NP; para esto necesitamos un proceso que modifique todos los componentes de un problema de tal manera que sean componentes equivalentes en el otro. Esta transformación debe preservar la información de tal manera cada vez que el primer problema da un resultado positivo realice el problema transformado, y cada vez que el primer problema da un resultado negativo sea el segundo problema.

Por ejemplo, una trayectoria de Hamilton en una gráfica, es aquella que visita todos los nodos exactamente una sola vez. Una gráfica no necesita ser completa para tener una trayectoria o circuito de Hamilton, podemos reducir el problema de un circuito de Hamilton al problema del agente viajero (problema de la clase NP de la siguiente manera:

- Cada nodo en la gráfica representa una ciudad.
- A cada arista en la gráfica, le asignamos el costo, de viajar entre las dos ciudades equivalentes, un valor de 1.
- Para cada par de nodos que no tienen arista conectándolos, asignamos el costo, de viajar entre las dos ciudades equivalentes, un valor de 2.

Esto convierte a la gráfica en un conjunto de ciudades interconectadas, ahora resolvemos el problema del agente viajero que se ha planteado: si se tiene un circuito de Hamilton en la gráfica, la solución que se obtiene será capaz de encontrar una solución viajando entre las ciudades que tienen un costo de 1. Si no se tiene un circuito de Hamilton, la solución del problema del agente viajero incluirá viajar entre al menos un par de ciudades con un costo de 2. Si tenemos n nodos en la gráfica, se tiene un circuito de Hamilton si la trayectoria del

agente viajero es de longitud n, y si no se tiene un circuito de Hamilton la trayectoria tendrá una longitud mayor a n.

3.4.2. Relaciones Básicas

Se observa que para todas las funciones de tiempo $t(n)$ y las funciones de espacio $s(n)$, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{NTIME}[t(n)]$ y $\mathsf{DSPACE}[s(n)] \subseteq \mathsf{NSPACE}[s(n)]$, debido a que una máquina determinista es un caso especial de una máquina no determinista. Además, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)] \subseteq \mathsf{NTIME}[t(n)] \subseteq \mathsf{NSPACE}[t(n)]$ debido a que en cada etapa, una máquina de Turing con k-cintas puede escribir sobre ella a lo más en $k = \mathcal{O}(1)$ celdas previamente sin escritura. A continuación se listan relaciones importantes, entre clases.

Sea $t(n)$ una función constructible-en-tiempo, y sea $s(n)$ una función constructible-en-espacio, $s(n) \geq \log n$.

1. $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DTIME}[2^{\mathcal{O}(t(n))}]$,
2. $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{\mathcal{O}(s(n))}]$,
3. $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$.
4. (**Teorema de Savitch**) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$.

Referente a las clases uniformes de complejidad de los circuitos lógicos, podemos observar que:

- La clase uACC^0 corresponde al cómputo en cualquier algebra con solución, por consiguiente esta clase es la subclase de uNC^1.
- En la clase uNC^0 todo bit de salida solamnte depende de $\mathcal{O}(1)$ bits de entrada, por consiguiente cualquier función en esta clase puede computarse por circuitos uAC^0 de profundidad dos, solamente utilizando una expansión en expresiones normalizadas DNF o CNF.

Por lo revisado en la sección 3.1.3, podemos ver que uNC^0 es extremadamente limitada, por ejemplo estos circuitos no pueden computar el *OR* lógico con una entrada de n bits, pero una de las sorpresas que nos encontramos al revisar su complejidad, consiste en que, a pesar de sus restricciones, la clase uNC^0 se puede considerar "cercana" en potencia computacional a la clase uAC^0, la cual conocemos es un subconjunto propio de la clase uACC^0, por consiguiente podemos plantear la siguiente relación entre estas clases:

$$\mathsf{uACC}^0 \subseteq \mathsf{uTC}^0 \subseteq \mathsf{uNC}^1$$

Recordemos que la u en estas clases se refiere a la uniformidad-$\mathsf{DLOGTIME}$, por ejemplo para uAC^0 representa a la clase de lenguajes que son reconocidos por una familia $(G_n)_{n\in\mathbb{N}}$ (toda entrada de tamaño n se evalúa por G_n compuertas) de circuitos lógicos construidos a partir

de compuertas *AND* y *OR* con un fan-in sin límite, tal que G_n tiene una profundidad constante y un tamaño polinomial en n y pueda construirse en DLOGTIME a partir de la cadena 1^n. uTC^0 tiene adicionalmente compuertas *MAJORITY* con fan-in sin limite. uNC^1 carece de compuertas *MAJORITY*, la profundidad de G_n es logarítmica en n, pero el fan-in esta acotado.

En la decada de los 1970, Nicholas Pippenger [ALR00] generaliza estas relaciones con tamaño polinomial y profundidad polilogaritmica, y propone el siguiente teorema que expresa las relaciones en cada nivel de la jerarquía definida por estas clases:

Teorema 3.4. *[ALR00, Teorema 3.4] Para toda $k \geq 0$,*

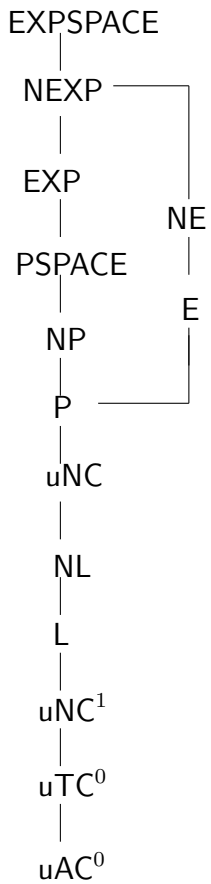
$$\mathsf{uNC}^k \subseteq \mathsf{uAC}^k \subseteq \mathsf{uTC}^k \subseteq \mathsf{uNC}^{k+1}.$$

En cuanto a las relaciones entre las clases de complejidad con base en los circuitos lógicos y las clases con base en la máquina de Turing tenemos:

Teorema 3.5. *[ALR00, Teorema 3.5]*

$$\mathsf{uNC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{uAC}^1.$$

La siguiente figura muestra las relaciones de inclusión que se tienen entre las clases canónicas de complejidad.



4. VALIDACIÓN DE DOCUMENTOS XML

El problema de validar un documento XML consiste en verificar, en una sola exploración y utilizando una cantidad de memoria fija, que su estructura cumpla con lo definido por un sistema de tipificado (en este estudio se trata de DTD); en función no del tamaño del documento XML sino de la información de tipificado. En otras palabras la validación la realiza un autómata de pila o PDA (Push-Down Automaton) mediante una exploración sobre un documento XML, a medida que va apareciendo en forma de un tren de caracteres. El problema tiene dos alternativas, en función de que la validación incluya o no la verificación de la buena conformidad, llamada *validación estricta* (strong validation). Si se verifica que se satisface la DTD asumiendo que la cadena de entrada corresponde a un documento XML bien formado, nos referimos simplemente a una *validación*. Podemos notar que la validación respecto a una DTD equivale a verificar la membresía del árbol asociado con el documento XML a un lenguaje regular de árbol , mientras que la validación por un autómata de estados finito o FSA (Finite State Automaton), equivale a la aceptación del árbol por una forma restringida de autómata de recorrido de árbol (tree-walking automaton).

4.1. Codificación de Documentos XML

4.1.1. La clase LogCFL

La clase LogCFL se define como el conjunto de lenguajes reducibles en LogSpace a lenguajes libres de contexto o CFL (Context-Free Languages), esta clase se caracteriza por tener una representación alterna con base en circuitos lógicos; también es conocida por la factibilidad de ser abordada desde la lógica como componente del enfoque descriptivo de la complejidad.

Definición 4.1. *Sean $A, B \subseteq \Sigma^*$. A es reducible de muchos a uno (many-one reducible) en* LogSpace *a B (denotdo como $A \leq_m^{\log} B$) si existe una función f computable con cota* LogSpace *tal que:*

$$\forall w \in \Sigma^*(w \in A \Leftrightarrow f(w) \in B).$$

Con estos elementos se puede definir la clase de complejidad LogCFL como:

Definición 4.2.

$$\text{LogCFL} = \{A \subseteq \Sigma^* : \exists B : CFL(A \leq_m^{\log} B)\}.$$

La clase LogCFL es una clase uniforme, es decir, todos los conjuntos en LogCFL son recursivos.

4.1.2. Complejidad y Lógica en la Validación XML

Para caracterizar las clases de complejidad en términos de una lógica, consideraremos cadenas como estructuras de primer orden cuyo universo consiste de un segmento inicial de los enteros $[1n]$, donde n es la longitud de la cadena y cada uno de estos enteros corresponde a una posición en la cadena. La FO sobre estas estructuras tiene acceso a un predicado unario $Q_a(i)$ para cada letra a en Σ, en el que la posición i de la cadena corresponde a una a, Un predicado binario $\leq$ que denota el orden usual sobre los enteros y un predicado binario $BIT(i, j)$ indicando que el $i^{ésimo}$ bit de j, en su representación binaria, tiene el valor de 1.

Immerman [Imm99] demuestra que las versiones no uniformes de FO y de CRAM[t(n)] (Concurrent Random Access Machine en tiempo paralelo $t(n)$) son iguales a AC, con base en esta relación se fundamenta el siguiente teorema:

Teorema 4.1. $\mathsf{FO} = \mathsf{uAC}^0$

Por otra parte consideramos a FOM:

Definición 4.3. FOM *es una extensión de* FO *que incorpora un cuantificador de mayoría M (majority) tal que* $Mx\varphi(x)$ *es verdadero cuando al menos la mitad de las posiciones* x *valida a* φ.

el mismo Immerman junto con Barrington y Straubing [BIS90] establecen el siguiente teorema:

Teorema 4.2. $\mathsf{FOM} = \mathsf{uTC}^0$

Los siguientes predicados son susceptibles de expresarse en FOM [BIS90]:

1. $x + y = z$
2. $x = \sharp y\varphi(y)$, es decir x es la cantidad de posiciones y de tal manera que $\varphi(y)$ se cumple.

Una función $f : \Sigma^* \rightarrow \Gamma^*$ para la cual existe una constante k tal que $|f(w)| \leq k|w|$ es una reducción FOM si existen las formulas FOM:

- $\varphi(x)$
- $\varphi_a(x)$

para todo símbolo $a \in \Gamma$ tal que, para toda estructura de primer orden de una palabra w y toda:

$i \in \{1, \cdots, k.|w|\}$, $\varphi(i)$ se cumple si y solo si:

$i = |f(w)|$ y $\varphi_a(i)$ se cumple si y sólo si la i-ésima posición de $f(w)$ es una a.

Con base a las consideraciones anteriores, ver [Loh01], tenemos que:

Lema 4.1. *[LaR96]*

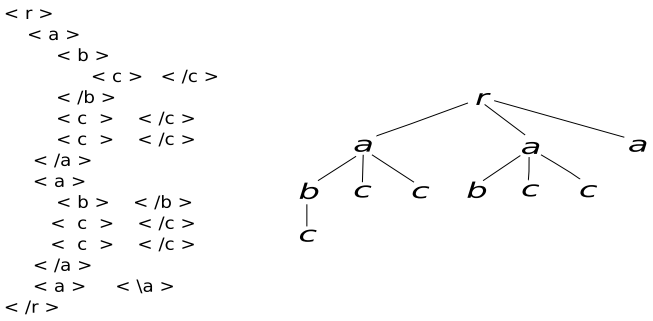


Fig. 4.1: Modelado de un documento XML

uTC^0 *y* uNC^1 *son cerrados bajo reducciones de* FOM.

En la codificación de cadenas un bosque es una cadena bien balanceada sobre $\Sigma \cup \overline{\Sigma}$. Esto se puede verificar con base en el siguiente lema.

Lema 4.2. *[BaC89]: Dada una palabra w sobre el alfabeto $\Sigma \cup \overline{\Sigma}$ es decidible en* uTC^0 *si w esta bien balanceado.*

Del lema anterior y de la relación $\mathsf{FO} = \mathsf{uAC}^0$ obtenemos la siguiente conclusión:

Existen formulas en FOM θ y θ^* tales que, para cualquier palabra w y cualquier posición i,j de la palabra:

$$w \models \theta^*(i, j)$$

si y sólo si $i < j$ y la subpalabra de w desde la posición i a la posición j esta bién balanceada y:

$$w \models \theta(i, j) \text{ si y solo } w \models \theta^*(i, j)$$

así mismo los paréntesis que cierran coinciden con el i^{esimo} caracter al j^{esimo} caracter.

4.1.3. Estructuras Arborecentes

Los documentos XML tienen una estructura en forma de árbol ordenado sin rango fijo, etiquetado con elementos en Σ.

Si tenemos un árbol t, la cantidad de sus nodos lo representamos como $|t|$. El ***rango*** de un nodo n es la cantidad de hijos del nodo. El ***rango*** de un árbol t es el máximo rango de n, sobre todos los nodos n de t.

Arboles como estructura de apuntadores. Este concepto de manera general corresponde a la codificación establecida por DOM (Document Object Model)[DOM08], ya que cada nodo es un objeto que contiene su etiqueta y sus apuntadores hacia:

1. su padre (Parent)
2. a sus hijos (Child)
3. a su hermano previo (Previous-Sibling)
4. a su siguiente hermano (Next-Sibling)

Si t es un árbol, el tamaño de la estructura de apuntadores asociado a t is aproximadamente a $4 \cdot |t|$ y por consiguiente también tiene como notación $|t|$.

Arboles como cadenas Una segunda codificación es la del modelo utilizado en la interfaz de SAX (Standard API for XML) [SAX08]. Los documentos XML se pueden codificar mediante la representación de un árbol en forma de cadena utilizando las marcas de apertura y de cierre para cada elemento. El procesamiento del documento en forma de cadena, ve una secuencia de marcas en el orden en que aparecen en el documento. Es la representación real del texto de un documento XML, sin valores de datos, lo cual se traduce en una sucesión de marcas de apertura y cierre. Corresponde a atravesar el árbol primero en profundidad (depth-first) primero a la izquierda (left-first). Aplicando la notación de [SeV02], tenemos que para cada $a \in \Sigma$, a representa ella misma la marca de apertura y $\overline{a}$ representa la marca de cierre para a. Sea $\overline{\Sigma} = \{\overline{a} | a \in \Sigma\}$. Con esta notación, la cadena asociada al árbol etiquetado del ejemplo anterior es:

$$r \quad a \quad b \quad c \quad \overline{c} \quad \overline{b} \quad c \quad \overline{c} \quad c \quad \overline{c} \quad \overline{a} \quad a \quad b \quad \overline{b} \quad c \quad \overline{c} \quad c \quad \overline{c} \quad \overline{a} \quad a \quad \overline{a} \quad \overline{r}$$

De una manera más general, asociamos a cada árbol etiquetado una representación en forma de cadena denotada como $[t]$ y se define inductivamente como sigue: si t es una raíz simple etiquetada como a, entonces $[t] = a\overline{a}$; si consiste en una raíz etiqueta a y subárboles $t_1 \dots t_k$ entonces $[t] = a[t_1] \dots [t_k]\overline{a}$. Si se trata de un conjunto de árboles, se denota por $L(T)$ el lenguaje consiste en las representaciones en cadena de los árboles en T.

Si t es un árbol, el tamaño de $[t]$ es dos veces la cantidad de nodos en t. Por consiguiente también utilizamos la notación $|t|$ para representar al tamaño de la cadena que codifica a t.

El siguiente lema nos da las complejidades de intercambiar estas dos representaciones.

Lema 4.3. *(i) Dado un árbol t como una estructura de apuntador, $[t]$ puede ser procesada en* LogSpace *a partir de t. (ii) Dado t como una estructura de cadena, la codificación de apuntadores se puede computar en* uTC^0 *a partir de $[t]$.*

Prueba: Con el propósito de cambiar de la codificación en cadena a la representación en apuntadores necesitamos calcular para cada nodo (i) su primer hijo (ii) sus hermanos a la izquierda y a la derecha (iii) su padre. Esto esta dado en LogSpace a partir de la representación de apuntador de t. Ya que $\mathsf{FOM} = \mathsf{uTC}^0$ podemos ver que esto se puede hacer en uTC^0 a partir de la representación de cadena de t y al obtener las formulas FOM correspondientes. A continuación identificamos un nodo de t con la posición de los paréntesis de apertura en su palabra codificada. Recordemos las definiciones de θ y θ^* como formulas FOM. Sea $Open(i)$ la fórmula FOM que verifica que la posición i esta en Σ (como opuesto a $\overline{\Sigma}$).

$Next - Sibling(i,j) \equiv Open(i) \vee Open(j) \vee i \leq j \vee \exists u(\theta(i,u) \vee u+1 = j)$
$Previous - Sibling(i,j) \equiv Open(i) \vee Open(j) \vee j = i+1$
$Parent(i,j) \equiv Open(i) \vee Open(j) \vee j \leq i \vee (i = j+1 \vee \theta^*(j+1, i-1))$
□

Notemos que la complejidad de *(i)* en el Lema anterior es óptima y que el cómputo de recorrer un árbol en *depth-first* es FLogSpace-hard bajo reducciones uNC^1, donde FLogSpace es la variante funcional de LogSpace.

4.2. Modelado de la Tipificación DTD

4.2.1. DTD como una Gramática Libre de Contexto

XML, en tanto que lenguaje de programación, puede describirse mediante una gramática libre de contexto o CFG (Context-Free Grammar) propia, la cual desempeña un papel relevante, debido a que forma parte del proceso de utilización del metalenguaje que realmente representa.

Definición 4.4. *Una gramática libre de contexto (CFG) es una cuarteta:*

$$G = \langle N, \Sigma, P, S \rangle$$

en la que:

- N *es un conjunto finito de símbolos no terminales,*
- P *es un conjunto finito de reglas de la forma* $A \overset{G}{\Rightarrow} \alpha$ *donde* $A \in N$ *y* $\alpha \in (N \cup \Sigma)^*$,
- $S \in N$ *corresponde al símbolo inicial.*

G genera una cadena $w \in \Sigma^*$ si existe una secuencia finita tal que:

$$S \overset{G}{\Rightarrow} \alpha_1 \overset{G}{\Rightarrow} \alpha_2 \overset{G}{\Rightarrow} \cdots \overset{G}{\Rightarrow} w.$$

Un conjunto $A \subseteq \Sigma^*$ es un lenguaje libre de contexto (CFL) si existe una gramática libre de contexto (CFG) G tal que:

$$\forall w \in \Sigma^*(w \in A \Leftrightarrow w \quad \text{es generada por } G).$$

El propósito de XML no es describir el formato de presentación del documento, sino que describe la semántica del texto que puede contener. Por ejemplo el texto «Vasconcelos, 309.» parece tratarse, entre varias opciones posibles a una dirección formada aparentemente por el nombre de una calle y el número de una ubicación, esto lo indica de la siguiente manera:

```
<DIRECCION> Vasconcelos, 309. </DIRECCION>
```

Sin embargo, no es obvio que <DIRECCION>signifique la dirección de un edificio, para aclarar lo que significan las etiquetas y que estructuras pueden aparecer entre ellas, los usuarios de cada especialidad con un interés común desrrollen estándares en forma de un DTD.

El lenguaje que describe una DTD es una notación para representar las variables y las producciones de una CFG, a continuación demostraremos que esta transformación es posible:

Teorema 4.3. *Una producción que incluye una expresión regular es posible transformarla a un conjunto de producciones válidas en una CFG.*

Prueba:

Paso Base: Si el cuerpo es una concatenación de elementos, la producción ya está en la forma admitida en la CFG.

Paso Inductivo: Si no es el caso base, tenemos cinco posibilidades, en función del operador final que se use.

1. La producción tiene la forma $A \to E_1, E_2$, donde E_1 y E_2 son expresiones permitidas en el lenguaje de la DTD. donde se utiliza el operador concatenación: introducimos dos variables nuevas dentro de la gramática, sean B y C; remplazamos la producción original por las producciones:
$$A \to BCB \to E_1C \to E_2$$

2. La producción tiene la forma $A \to E_1|E_2$. Para el operador unión podemos transformar la producción en el par siguiente de producciones:
$$A \to E_1A \to E_2$$

3. La producción tiene la forma $A \to (E_1)^*$. Introducimos una nueva variable B y reemplazamos esta producción por:
$$A \to BAA \to \epsilon B \to E_1$$

4. La producción tiene la forma $A \to (E_1)^+$. Introducimos una nueva variable B y reemplazamos esta producción por:
$$A \to BAA \to BB \to E_1$$

5. La producción tiene la forma $A \rightarrow (E_1)?$. Reemplazamos esta producción por:

$$A \rightarrow \epsilon A \rightarrow E_1$$

□

Por ejemplo consideremos el siguiente documento XML:

```
<?xml version="1.0"?>
<!DOCTYPE LIBRO [
    <!ELEMENT LIBRO (P*)>
    <!ELEMENT P (#PCDATA)>
]>
<LIBRO>
    <P>Capítulo 1 - Introducción</P>
    <P>Capítulo 2 - Conclusión</P>
    <P>Index</P>
</LIBRO>
```

La línea del DTD que contiene una expresión regular:

```
 <!ELEMENT LIBRO (P+)>
```

es equivalente a las producciones:

$$Libro \rightarrow AA \quad \rightarrow P^+$$

La última de estas producciones (P+) no tiene una fórmula válida en una CFG, por consiguiente introducimos una nueva producción:

$$Libro \rightarrow AA \quad \rightarrow P \quad A \quad | \quad P$$

4.2.2. DTD y Automatas de Árbol

Consideramos que un autómata de estados finito no determinista o NFA (No-determiniatic Finite State Automaton) y un autómata de estados finito determinista o DFA (Determiniatic Finite State Automaton) son autómatas sin transiciones ϵ y que admiten cadenas de caracteres. Si A es un NFA, e una expresión regular, $L(A)$ y $L(e)$ denotan a los lenguajes regulares asociados. Las gramáticas libre de contexto (CFG) y los lenguajes libres de contexto (CFL) están definidos de acuerdo a lo establecido en la sección anterior, si se trata de una CFG, $L(G)$ denota al CFL asociado.

Dado que los DTDs y sus variantes proporcionan un mecanismo de tipificado para documentos XML, usaremos dos perspectivas de tipificación para árboles:

- La primera corresponde a la propuesta por el W3C para documentos XML. Esta DTD constituye una gramática libre de contexto extendida, es decir que los lados derechos de las producciones son expresiones regulares que admiten terminales y no terminales sobre un alfabeto Σ . Ya que las expresiones regulares son cerradas bajo la unión, podemos asumir que cada una de las DTD tienen una regla única $a \to L$ para cada símbolo $a \in \Sigma$, denotamos a la expresión regular L para a por R_a. Un documento XML satisface a la DTD τ, o es válido respecto a τ, si para cada nodo etiquetado a, la lista ordenada de sus hijos forman una palabra contenida en R_a. Por ejemplo, el árbol etiquetado de la figura 4.1 es válido para la DTD siguiente:

  $r \to a^*$
  $a \to bc^*|\epsilon$
  $b \to c|\epsilon$
  $c \to \epsilon$

- El segundo sistema de tipificado corresponde al utilizado por *XML-Schema*, el cual esta constituido por un DTD extendido (EDTD) con *especializacion* (también llamada tipos desacoplados), el cual, intuitivamente, permite definir el tipo de una marca para varios casos en función del contexto.

Definición 4.5. *Un EDTD sobre Σ es una tupla $\tau = (\Sigma, \Sigma', \tau', \mu)$ donde (i) Σ y Σ' son alfabetos finitos, (ii) τ' es una DTD sobre Σ'; y (iii) μ es un mapeo de Σ a Σ'.*

Un documento con estructura de árbol t sobre Σ satisface un EDTD τ, si $t \in \mu(SAT(\tau'))$

Intuitivamente, Σ' proporciona para algunas a's en Σ un conjunto de especializaciones de a, concretamente aquellas $a' \in \Sigma'$ para las cuales $\mu(a') = a$. También μ representa el homomorfismo inducido por μ sobre las cadenas y árboles, extendido cuando se necesita para símbolos en $\overline{\Sigma}'$ mediante $\mu(\overline{a}') = \overline{\mu(a')}$.

Cuando consideramos solamente representaciones de cadena puede ser más fácil ver a DTD y EDTD como gramáticas libres de contexto extendidas con alfabeto de terminales $\Sigma \cup \overline{\Sigma}$, alfabeto de no terminales $V = \Sigma'$, que contiene reglas de la forma $S \to aR_S\overline{a}$, donde $S \in V$, $a \in \Sigma$, R_S es una relación regular sobre V. Las DTDs tienen la restricción que para cada $a \in \Sigma$ hay una sola regla que involucra a a. Las EDTDs no tienen esta restricción y tienen una regla por elemento en Σ'.

Si τ es un (E)DTD, SAT(τ) es el conjunto (regular) de árboles que satisfacen τ y $L(\tau)$ es el lenguaje sobre $\Sigma \cup \overline{\Sigma}$ que consiste en las representaciones de cadena de todos los documentos de árbol en $SAT(\tau)$, esto es $\{[t]|t \in SAT(\tau)\}$.

En el caso de un autómata de árbol (TA). la definición para los árboles con rango variable (unranked) es como sigue, pueden ser desde-abajo-hasta-arriba (bottom-up, BU) o desde-arriba-hasta-abajo (top-down, TD) y ya sea determinista (D) o no determinista (N).

Un autómata de árbol no determinista desde-arriba-hasta-abajo (non-deterministic top-down tree automata, NTDTA) es una tupla (Σ, Q, q_0, δ) en la que Q es un conjunto de estados finito, q_0 es el estado inicial, y $\delta \subseteq \Sigma \times Q \times Q^*$ es tal que $\delta(a, q)$ es un lenguaje regular sobre Q para toda $a \in \Sigma$ y cada $q \in Q$ (asumimos que para $a \in \Sigma$ y cada $q \in Q$, $\delta(a, q)$ está dada como un

NFA). El procesamiento comienza en la raíz del árbol, en el estado inicial, y continua hacia las hojas del árbol asignando estados a los hijos del nodo de acuerdo a su etiqueta, α, el estado actual, q, y a $\delta(\alpha, q)$. El árbol se acepta si, para cada hoja n del árbol, el autómata alcanza n en un estado q tal que ϵ esta en $\delta(etiqueta(n), q)$. Un TDTA determinista (DTDTA) es un NTDTA en el que para toda $a \in \Sigma$ y cada $q \in Q$ y cada $n \in \mathbb{N}$ existe una $w \in \delta(a, q)$ única de longitud n.

Un autómata de árbol no determinista desde-abajo-hacia-arriba o NBUTA (Non deterministic Bottom-Up Tree Automata) es una tupla (Σ, Q, F, δ) en la que Q es un conjunto finito de estados, F es un conjunto de estados de aceptación y $\delta \subseteq \Sigma \times Q^* \times Q$ en el cual $\{w \in \Sigma^* | \delta(a, w, q)\}$ es un lenguaje regular sobre Q para cada $a \in \Sigma$ y toda $q \in Q$. El procesamiento inicia en las hojas, al asignarles a cada una de las hojas de etiqueta α un estado de acuerdo a $\delta(\alpha, \epsilon)$ y prosigue hacia la raiz del árbol asignando estados a un nodo de acuerdo a su etiqueta, a, la secuencia de estados de sus hijos, w, y $\delta(a, w)$. El árbol se acepta si el autómata alcanza la raíz en un estado de aceptación. Un BUTA determinista (DBUTA) es un NBUTA en el que cada $a \in \Sigma$ y toda $w \in Q^*$ existe una q única en la que $\delta(a, w) = q$.

Si A es un autómata de árbol (desde-abajo-hacia-arriba o cima-hacia-abajo) denotaremos por SAT(A) al conjunto de árboles aceptados por A, y por $L(A)$ al conjunto de cadenas de codificación de los árboles aceptados por A. Sea T un conjunto de árboles con rango variable y etiquetado. T es aceptado por un EDTD si y solo sí es reconocido por un NBUTA si y sólo si T es reconocido por un NTDTA sí y sólo sí T is reconocido por un DBUTA. En este caso decimos que se trata de un lenguaje de árbol regular. Como se muestra en [CD+07], tanto un DTD como un DTDTA fallan al capturar todos los lenguajes regulares.

Finalmente consideramos un autómata de recorrido-de-árbol (tree-walking automata, TWA). Un TWA no determinista (NTWA) es una tupla (Σ, Q, F, δ) en la que Q es un conjunto finito de estados, q_0 es el estado inicial, F es un conjunto de estados de aceptación y $\delta \subseteq \Sigma \times Q \times \{raíz, hoja, hijo{-}izquierdo, hijo{-}derecho, otros\} \times Q \times \{arriba, abajo, izquierda, derecha, sin{-}movimiento\}$. El procesamiento comienza en la raíz en el estado inicial y continua el recorrido del árbol de acuerdo a: cuando alcanza el nodo n con etiqueta a en el estado q, el autómata verifica si el estatus del nodo actual es una raíz o una hoja o un hijo en el extremo izquierdo (o en el extremo derecho) o solamente un nodo interior, y de acuerdo a $\delta(a, q,$,status) cambia su estado a q' y se mueve hacia arriba, hacia abajo al hijo en el extremo izquierdo, hacia el siguiente hermano, hacia el hermano anterior, o se queda sin moverse en el nodo actual del árbol. Un TWA determinista (DTWA) es un NTWA en el que $\delta(a, q,$status) es único para cada $a \in \Sigma$ y para cada $q \in Q$. Se puede demostrar que NTWA y DTWA reconocen solamente lenguajes de árbol regulares. Todavía se tiene como problema abierto la demostración de si un NTWA reconoce todos los lenguajes de árbol regulares.

4.3. Complejidad de la Validación

4.3.1. Complejidad de los Datos

En está sección se analiza la complejidad de los datos del problema de la validación para los sistemas de tipificado. Como se mencionó anteriormente estos sistemas definen un lenguaje de árbol regular. Por consiguiente empezamos con el caso más general y a considerar la membrecía de un árbol en un lenguaje de árbol regular. Esto dará las cotas superiores (upper-bounds) (en función del código seleccionado para el árbol de entrada) para los sistemas de tipificación que se han considerado. En la última parte de esta sección veremos que estas cotas superiores realmente coinciden para todos ellos. Por consiguiente la complejidad de los datos en el problema de validación no depende de la tipificación utilizada.

Sea T un lenguaje de árbol regular. Consideremos el siguiente problema de decisión [Seg03]:

$[VAL_T]$: ENTRADA: un árbol etiquetado
SALIDA: verdadero sí y solo sí $t \in T$.

La complejidad de VAL_T depende de la codificación utilizada por t. Consideremos dos codificaciones para los árboles: el correspondiente al usado en DOM y al utilizado en SAX, ambos han sido analizados. Empezamos con el caso en el que t se codifica como estructura (en árbol) de apuntadores como mejor se ilustra la diferencia entre árboles con rango fijo (ranked) con aquellos que carecen de rango fijo (unranked).

A t lo consideramos como una estructura de apuntadores, necesitamos verificar si $t \in \mathrm{SAT(A)}$ para algún autómata de árbol con rango variable A. Procedemos como sigue: primero consideramos el caso en el que todos los árboles tienen un rango fijo y a continuación reduciremos el caso de rango variable (unranked) a un caso con rango fijo (ranked).

Asumiendo inicialmente que todos los árboles tienen un rango fijo acotado por k. Sea T un lenguaje de árbol regular (con rango k) arbitrario y a A como un DBUTA que reconoce a T. Sea Q el conjunto de estados de A y Γ sea $(Q \cup \{\sqcup\})^k$, donde $\sqcup$ corresponde a un nuevo símbolo.

A continuación, i representa un apuntador sobre t y S es una pila sobre el alfabeto Γ. Si $H \in \Gamma$ y $q \in Q$, $ACTUALIZA(H, q)$ es la función que remplaza al símbolo $\sqcup$ más a la izquierda de H por q. Sea $<$ un orden total sobre los nodos de t y $NEXT-SIBLING < (i)$ y $FIRST-CHILD < (i)$ sean las funciones que regresan los apuntadores hacia los nodos con correspondencia al próximo-hermano (primer-hijo) del nodo apuntado por i en función de $<$. Inicialmente i apunta a la raíz de t y S está vacío.

Consideremos el siguiente algoritmo secuencial mediante el cual un autómata de árbol procesa un árbol [Seg03]:

mientras verdadero **hacer** {
  si i es una *hoja* **hacer** {
    /* si i es el primero de los hermanos que se evalúa entonces empezamos un nuevo cómputo, si no llamamos a los cómputos previos de la pila */
    si $i := primer - hijo_<$ **hacer** $H := \sqcup^k$

si no $H := pop$; **fsi**
$q := \delta(label(i), \epsilon)$; $H := actualiza(H, q)$;
mientras i no tenga $siguiente - hermano_<$ **hacer** {
/* Si todos los hermanos de i ya han sido evaluados podemos computar el estado del $padre(i)$ usando H y continuar hacia arriba*/
si i es la raíz devolver q;**fsi**
$i := padre(i)$; $q := \delta(i, H)$;
si i es un $primer - hijo_<$ **hacer** $H := \sqcup^k$
si no $H := pop$ **fsi**;
$H := actualiza(H, q)$;
} **fmientras**/*fin i no tiene $siguiente - hermano_<$*/
/* i tiene un hermano que todavía no sido evaluado, asi que almacenamos la evaluación actual en la pila y continuamos */
$push(H)$; $i := siguiente - hermano_<(i)$;
}/*fin del hacer si i es una hoja*/
/* exploración $Profundidad - primero < -primero$ del árbol */
si no $i := primer - hijo_<(i)$ **fsi** /*fin si inicial*/
}**fmientras** /*fin mientras inicial*/

Este algoritmo particiona al árbol de la manera *profundidad-primero <-primero*, computando para cada nodo, los estados alcanzados por el autómata en ese nodo. Este estado puede obtenerse solamente cuando todos sus hijos se han procesado y la pila S se utiliza para almacenar los cómputos parciales. El análisis de los recursos que necesita el algoritmo depende de $<$. La aproximación inicial se deberá hacer usando el órden *profundidad-primero izquierda-primero* de t, (recordemos que los hermanos están ordenados) para $<$. De esta manera se puede ver que las funciones $NEXT - SIBLING_<$ y $FIRST - CHILD_<$ son computables en LogSpace, pero notemos que un elemento se introduce (pushed) en la pila S, cada vez que un *primer-hijo* de un *siguiente-hermano* se visita. Por consiguiente la profundidad de la pila S, puede ser lineal respecto al tamaño de t, y el espacio general utilizado es $O(k.|t|)$. Una mejor aproximación consiste en ordenar a los hermanos de acuerdo al tamaño de sus correspondientes subarboles, tomando en primer lugar al hermano más grande y más a la izquierda en el caso de que sean iguales (esta estrategia de considerar primero los subárboles más grandes es común en el procesamiento de árboles).

Referente al caso en que se visita un *primer-hijo* de un *hermano-siguiente* el tamaño del subárbol correspondiente se divide cuando menos entre 2. Por consiguiente la profundidad de la pila S esta ahora acotada por $\log |t|$. Notemos que el tamaño del subárbol se puede computar en LogSpace y por consiguiente las funciones $NEXT - SIBLING_<$ y $FIRST - CHILD_<$ permanecen computables en LogSpace. La complejidad en espacio es por consiguiente $\mathcal{O}(k. \log |t|)$ el cual está en LogSpace cuando k es fijo. Por lo tanto tenemos:

Lema 4.4. *[GK+05] Si T representa un lenguaje regular de árbol con rango fijo entonces Val_T está en* LogSpace.

Prueba: Este algoritmo no generaliza a los árboles con rango variable (unranked tree) debido

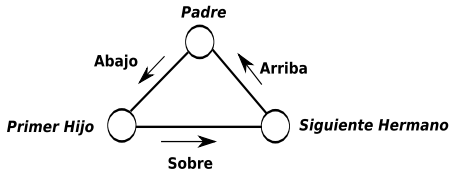


Fig. 4.2: Algoritmo de Travesía de un Árbol en LOGSPACE

a que es lineal en el rango del árbol el cual, en general, puede ser lineal en $|t|$. Por esta razón, cuando los árboles no tienen rango fijo, y la evaluación primero-profundidad <-primero está condenado a fallar. Con el propósito de mantener la complejidad LogSpace, la travesía del árbol se hace utilizando el siguiente órden: el sucesor j de un nodo i es ya sea que el hijo más a la izquierda de i o bien sea su próximo-hermano en función de que sea o no el tamaño del subárbol con raíz en i más grande que el tamaño total de los subárboles con raíz en sus hermanos a la derecha. Se puede mostrar que el algoritmo para el caso del rango adaptado para que la travesía esta en LogSpace.

Otra forma de obtener la complejidad LogSpace es pre-procesar al árbol con rango variable para codificarlo como un árbol con rango fijo, esta reducción es muy común en la literatura algorítmica. Sea $\sharp$ un nuevo símbolo y $\Sigma^\sharp$ sea $\Sigma \cup \{\sharp\}$. La transformación toma un árbol etiquetado con rango variable t sobre Σ y nos proporciona un árbol binario etiquetado t' sobre $\Sigma^\sharp$. La transformación se denota por $RANGO$ y se define inductivamente como sigue (ver Figura 4.3): Si t es un árbol vacio entonces $RANGO(t)$ es un nodo simple etiquetado con $\sharp$. Sean $\overline{T}$ y $\overline{T}$' los bosques posiblemente vacios y t sea un árbol cuya raíz está etiquetado por $a \in \Sigma$ y su hijo está en el bosque $\overline{T}$, entonces $RANGO(t\overline{T}')$ es el árbol binario con raíz en un nodo etiquetado por a cuyo hijo izquierdo es $RANGO(\overline{T})$ y su hijo derecho es $RANGO(\overline{T}')$,

Se puede ver que $RANGO$ es 1-1 y es particularmente importante que preserve su regularidad:

Lema 4.5. *[GK+05] Dado un lenguaje regular de árbol T el conjunto $T = \{t' | t' = RANGO(t)\}$ es también un lenguaje de árbol regular.*

Esta codificación puede realizarse en LogSpace, el algoritmo copia los nodos de entrada, posiciona apropiadamente su enlace (esto puede hacerse en LogSpace ya que solamente los nodos locales están involucrados), y finalmente agrega tantos nodos $\sharp$ como se requiera.

Lema 4.6. *[GK+05] Sea un árbol codificado por su estructura de apuntadores. Entonces la estructura de apuntadores que codifica $RANGO$ puede calcularse en* LogSpace *desde t.*

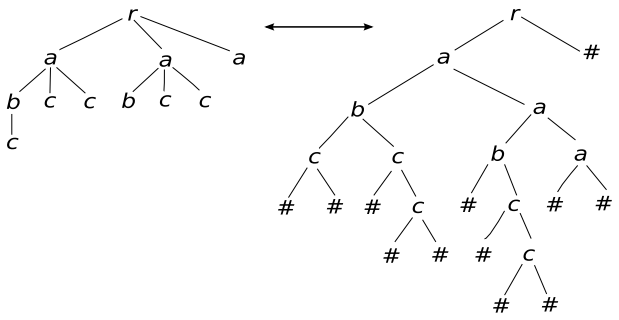


Fig. 4.3: De árbol con rango variable (unranked) a árbol con rango fijo

A partir de los tres lemas anteriores se plantea el siguiente resultado:

Teorema 4.4. *[Seg03] Para todo lenguaje de árbol regular, si esta codificado como una estructura de apuntadores, Val_T esta en* LogSpace.

La complejidad del teorema anterior es la mejor posible, como se muestra a continuación.

Teorema 4.5. *[Seg03] Existe T tal que, asumiendo que esta codificado como una estructura de apuntadores, Val_T es completo para* LogSpace *bajo reducciones* uAC^0.

Prueba: Sea T la familia regular de árboles que consiste en una sola rama que forma una cadena en el lenguaje regular $a^*ba^*ca^*$. Probamos la dificultad de Val_T al reducirla al siguiente problema LogSpace completo bajo reducciones uAC^0

ORD: ENTRADA una gráfica dirigida $G = (V, E)$ la cual es una línea y dos nodos v_i, v_j
SALIDA: verdad sí y solo sí $v_i < v_j$ en el órden inducida por la gráfica.

Sea G una gráfica dirigida que es una línea y v_i, v_j, dos nodos de G. G puede verse como un árbol t y una estructura de apuntadores para t puede construirse a partir de G de la forma obvia. Las etiquetas de los nodos de t están puestas de tal manera que v_i esta etiquetado con b, v_j esta etiquetado con c y los otros nodos están etiquetados con a. Esta reducción es de primer orden, es decir FO, y por consiguiente computable en uAC^0.

Se puede verificar que ORD es verdadero para G, v_i, v_j si y sólo si $t \in T$. Esto concluye la prueba. □

Si el árbol t esta codificado como una cadena, y es entrada de Val_T, recordemos que $[t]$ denota precisamente su codificación en cadena, el resultado que se puede plantear es el siguiente:

Teorema 4.6. *[Seg03] Para todo T, si t esta codificado como una cadena, el algoritmo Val_T esta en* uNC^1.

Prueba: Siguiendo la misma secuencia que en la prueba anterior, como primer paso reducimos el caso de rango variable a uno de rango fijo. La reducción está basada en el siguiente Lema:

Lema 4.7. *[Seg03] Sea t un árbol $[RANGO(t)]$ computable en* uTC^0 *a partir de $[t]$.*

Prueba. Ya que $\mathsf{FOM} = \mathsf{uTC}^0$ (ver sección 4.2) es suficiente que demos una formula lógica FOM que corresponda a la reducción. Si S es un subconjunto de Σ denotamos con $Q_S(i)$ al predicado $\bigvee_{a\in S} Q_a(i)$. Como hemos visto, dada una posición i que contiene un símbolo de Σ la única posición j que contiene los paréntesis de cierre es la única j verificando la fórmula FOM $\theta(i,j)$

A continuación identificamos un nodo del árbol por la posición de los paréntisis de apertura correspondiente en su palabra codificada. La siguiente formula FOM por lo tanto verifica que un nodo i es una hoja (Leaf) de t.

$$Leaf(i) \equiv Q_\Sigma(i) \wedge \theta(i, i+1)$$

La siguiente formula FOM verifica que un nodo i no tenga hermano derecho (NoRightSibling).

$$NoRightSibling(i) \equiv \exists j(\theta(i,j) \wedge (Q_{\overline{\Sigma}}(j+1) \vee Last(j)))$$

La siguiente fórmula verifica que dos nodos i y j son hermanos (Sibling).

$$\begin{aligned} Sibling(i,j) \equiv \quad & (i \leq j \wedge \exists u, v(\theta(i,u) \wedge \theta(j,v) \wedge (u+1 = j \vee \\ & \theta^*(u+1, j-1))) \vee (j \leq i \wedge \exists u, v(\theta(i,u) \wedge \\ & \theta(j,v) \wedge (v+1 = i \vee \theta^*(v+1, i-1)))) \end{aligned}$$

Finalmente la siguiente fórmula FOM verifica que j es un ancestro (Ancestor) de i.

$$Ancestor(i,j) \equiv \exists u(j \leq i \leq u \wedge \theta(j,u))$$

Sea $t' = RANGO(t)$. Por definición t' contiene a lo más dos nodos extras por nodo de t, por lo tanto $|RANGO(t)| \leq 3.|t|$. De una manera más precisa t' contiene los mismos nodos que t más los nodos etiquetados con $\sharp$, uno para cada hoja de t y uno para cada nodo de t que no tiene hermano derecho. Debido a que cada nodo del árbol corresponde a 2 caracteres en su representación de cadena el tamaño z de $[t']$ se puede expresar por la siguiente fórmula FOM (recordemos que los siguientes predicados se pueden expresar en FOM : $x + y = z$ y $x = \sharp$ y $\varphi(y)$, el número de posiciones y tal que $\varphi(y)$):

$$\exists x, y(z = max + 2x + 2y \wedge x = \sharp \quad iLeaf(i) \wedge y = \sharp i \quad NoRightSibling(i))$$

Sea n un nodo etiquetado por $a \in \Sigma$ en t. Sea i la posición en $w' = [t']$ de la marca de apertura de n y j sea su posición en $w = [t]$. Buscamos una fórmula FOM que defina i a partir de j y de w. i se computa contando la cantidad de marcas de cada tipo que ocurre antes de i en

w'. Notemos primero que la travesía *profundidad-primero izquierda-primero* de t y t' pasará por los mismos nodos en el mismo orden (asumimos que nos brincamos los nodos etiquetados por $\sharp$ en t'). Por consiguiente antes de i (en w') se tienen tantas marcas de apertura como las que tenemos antes de j (en w). Esta cantidad se denota por x en la fórmula siguiente. Antes de i encontramos tantas $\sharp$ como tantos nodos tenemos antes de j los cuales son ya sea una hoja o carecen de hermano derecho, esta cantidad está representada por y en la siguiente fórmula, notemos que cada $\sharp$ esta seguida inmediatamente por un $\overline{\sharp}$ y por lo tanto cuenta por dos caracteres. De esta manera, con el objeto de obtener i a partir de j nos queda computar la cantidad de marcas de cerrado que deberán presentarse antes de i. Para realizar esto, notemos que es suficiente computar la cantidad de marcas de cerrado antes de j y quitar todas aquellas que tienen un hermano que sea ancestro de n y el cual esta por consiguiente todavía no completamente procesado por $RANGO$. Esta cantidad se representa por z en la siguiente fórmula. Concluimos entonces que la siguiente fórmula FOM $\varphi_a(i)$ proporciona las posiciones en t' etiquetadas por a:

$$\begin{array}{l}
\exists j(Q_a(j) \wedge \\
x = \sharp u \quad u \leq j \wedge Q_\Sigma(u) \wedge \\
y = \sharp u \quad \exists v(u \leq v \leq j \wedge \theta(u,v)) \wedge (Leaf(u) \vee NoRightSibling(u)) \wedge \\
z = \sharp u \quad \exists v(u \leq j \wedge Q_{\overline{\Sigma}}(u) \wedge \theta(v,u) \wedge \\
\neg(\exists w(Sibling(v,w) \wedge Ancestor(w,j)))) \wedge \\
i = x + 2y + z)
\end{array}$$

A continuación consideremos i como la posición en w' de la marca de cierre de n, y a j como su posición en w. De nuevo necesitamos computar i a partir de j y de w. La contabilidad de las marcas de apertura y de los símbolos $\sharp$ se hace siguiendo el mismo camino anterior. Para contar las marcas de cerrado, notemos que en w' la marca de cierre de n ocurre cuando el follaje completo de los hijos de n este procesado excepto para las marcas de cierre de los hermanos a la izquierda de n. Por consiguiente la fórmula FOM $\varphi_{\overline{a}}(i)$ siguiente nos proporciona las posiciones en w' etiquetadas por $\overline{a}$ (en esta misma fórmula, j' representa la posición de las marcas de apertura de n y k la posición de la marca de cierre del hermano más a la derecha de n. Esto se ejecuta por la primera línea de la fórmula):

$$\begin{array}{l}
\exists j, j', k \quad (Q_{\overline{a}}(j) \wedge \theta(j', j) \wedge \theta^*(j', k) \wedge \\
(\forall k' \quad k < k' \rightarrow \neg\theta^*(j', k')) \wedge \\
x = \sharp u \quad u \leq k \wedge Q_\Sigma(u) \wedge \\
y = \sharp u \quad \exists v(u \leq v \leq k \wedge \theta(u,v)) \wedge (Leaf(u) \vee NoRightSibling(u)) \wedge \\
z = \sharp u \quad \exists v(u \leq k \wedge Q_{\overline{\Sigma}}(u) \wedge \theta(v,u) \wedge (\neg Sibling(v,j) \vee j \leq u)) \wedge \\
i = x + 2y + z)
\end{array}$$

Las fórmulas para $\sharp$ y $\overline{\sharp}$ se obtienen siguiendo el mismo procedimiento. Es suficiente suponer que el nodo n en t es ya sea una hoja o carece de hijo derecho, entonces para una hoja, la posición actual de la marca de apertura de n en w' se procesa como se hizo anteriormente y se agrega 1 para $\sharp$, 2 para $\overline{\sharp}$. Si n no tiene hijo izquierdo la posición de la marca de cierre de n en wt$'$ es computado como se hizo anteriormente, eliminando 1 para $\overline{\sharp}$ y a 2 for $\sharp$.□

El caso con rango fijo se prueba adaptando las técnicas utilizadas por Lohrey [Loh01] para esta nueva codificación, lo cual lo reduce, en uTC^0, al problema de verificar la pertenencia a

un CFL de paréntesis, y esto nos lleva a un procesamiento que está en uNC^1

Con este antecedente planteado, se puede establecer lo siguiente:

Teorema 4.7. *[Seg03] Para todo T, si t es de rango fijo y esta codificado como una cadena, el algoritmo Val_T esta en uNC^1.*

Prueba. Sea T un lenguaje regular de árbol, y T' el lenguaje regular de árbol que acepta la imagen de T para $RANGO$ Sea $A = (\Sigma^\sharp, Q, \delta, q_0, F)$ un BUTA que reconoce a T'. Sea $G = (Q, \Sigma^\sharp, R, q_0)$ la CFG en la que R contiene $q \to aq'q''\overline{a}$ si y sólo si $\delta(a, q', q'') = q$. Se puede ver que $t \in T$ si y sólo si $[RANGO(t)]$. Por consiguiente con base al lema 4.7 y sabiendo que uTC^0 y uNC^1 son cerrados por reducciones FOM [Loh01], es suficiente mostrar que $[RANGO(t)] \in L(G)$ se puede verificar en uNC^1. Esto se puede lograr adaptando la prueba de [Loh01] a nuestra codificación de un árbol. Más adelante transformamos $t' = RANGO(t)$. Sea β la operación de cadena definida inductivamente en una representación de cadena de árboles, los árboles son ahora binarios y completos, como sigue:

$$\beta(\epsilon) = \epsilon$$
$$\beta(auv\overline{a}) = (a\beta(u)\beta(v))$$

donde u y v son cadenas balanceadas.

Consideremos ahora la CFG:

$$G' = (Q, \Sigma^\sharp \cup \{(,)\}, R', q_0)$$

donde R' contiene a $q \to (aq'q'')$ si y sólo si:

$$q \quad \to \quad a \quad q' \quad q'' \quad \overline{a}$$

esta en R. Es posible verificar que $[t] \in L(G)$ si y sólo si $\beta([t]) \in L(G')$, así mismo se puede mostrar que β es una reducción FOM.

Sea $w = [t]$ para un árbol binario completo. La fórmula FOM que da el tamaño i de $w' = \beta(w)$ esta dado por:

$$i \quad = \quad 3.\sharp j \quad Q_{\Sigma^\sharp}(j)$$

La fórmula FOM de las posiciones i de w' que contiene el símbolo $a \in \Sigma^\sharp$ están dadas por la fórmula siguiente. En esta fórmula j propone la posición en w del nodo cuya imagen para β es i, x computa la cantidad de nodos que ya han sido procesados completamente por β antes de alcanzar a j (notemos que cada uno de estos nodos esta codificado con 3 caracteres en w' y todos ellos están antes de i) e y computa los nodos que todavía no han sido procesados completamente por β antes de alcanzar j, concretamente los ancestros de j (notemos que cada uno de estos nodo tiene exactamente 2 caracteres que antes de i y uno después del paréntesis de cierre).

$$\begin{aligned}\varphi_a(i) \quad \equiv \quad & \exists j Q_a(j) \wedge \\ & x = \sharp u \exists v Q_\Sigma(u) \wedge u \leq v \leq j \wedge \theta(u,v) \wedge \\ & y = \sharp u Ancestor(u,j) \wedge \\ & i = 3x + 2y\end{aligned}$$

Las formulas para "(" y para ")" se obtienen de una manera similar. Recordemos que uTC^0 y uNC^1 son cerrados bajo reducciones de FOM; por consiguiente esto es suficiente para mostrar que $w \in L(G')$ puede ser verificado en uNC^1. En general verificar que una palabra pertenece a un CFL esta en LogCFL el cual se cree que es mucho más poderoso que uNC^1. G' representa una CFG especial, la cual consiste en una CFG de paréntisis como se define en [McN67]. Esto resulta en que esa pertenencia a un CFL de paréntisis puede ejecutarse en uNC^1. Esto concluye la prueba del teorema. □

La cota superior del Teorema 4.6 es la mejor posible debido a que, existen lenguajes regulares de cadena L, tal que el problema de verificar si una palabra w esta en L es completo para uNC^1 bajo reducciones LogTime. Tal lenguaje puede encontrarse en [BIS90]. Este lenguaje regular puede convertirse en un lenguaje regular de árbol, por consiguiente tenemos:

Teorema 4.8. *[GK+05] Existe T tal que Val_T* es completo para uNC^1 *bajo reducciones* DLogTime.

Todos los sistemas de tipificado mencionados, TA, TWA, DTD, EDTD, definen leguajes regulares de árbol. Además, todos ellos pueden codificar cualquier lenguaje regular de cadena y existen lenguajes regulares para los cuales probar la pertenencia esta en $\mathsf{uNC} - completo$ (ver [Str94])cuando la entrada es una cadena, y en LogSpace-completo cuando la entrada esta dada como una estructura de apuntadores. Con base en los teoremas 4.4 y 4.6 obtenemos la siguiente conclusión:

Sea τ un DTD, EDTD, TWA, o un TA. Si el árbol de entrada esta codificado como una estructura de apuntadores la complejidad de Val_τ esta en LogSpace*, si esta codificado con una cadena, la complejidad de Val_T esta en uNC^1. Estas cotas superiores constituyen la óptima para cada uno de los sistemas de tipificado.*

4.3.2. Complejidad Combinada

En la sección anterior la entrada consistía simplemente en la codificación de un árbol, en esta sección consideramos el problema de validación cuando la tipificación es también parte de la entrada. Sea C una clase de tipificado de árboles. Por ejemplo puede ser la clase de todos los DTDs, o todos los EDTDs o todos los autómatas de árbol, etc. Más formalmente planteamos los siguientes problemas:

$[ArbolM(C)]$ ENTRADA: un árbol t, una tipificación $\tau \in C$
SALIDA: es verdad si y sólo sí $t \in SAT(\tau)$

Debido a que todas las tipificaciones que hemos considerado constituyen una generalización de los lenguajes regulares de cadena, es conveniente revisar la complejidad en el procesamiento de cadenas.

[$CadenaM(C)$] ENTRADA: una cadena w, una tipificación $\tau \in C$
SALIDA: es verdad si y sólo sí $w \in \mathcal{L}(\tau)$

Lema 4.8. *1. CadenaM(DFA) es* LogSpace*-completo bajo reducciones* uAC^0.

2. CadenaM(NFA) es NLogSpace*-completo bajo reducciones* uAC^0.

3. CadenaM(Expresión Regular) es NLogSpace*-completo bajo reducciones* uAC^0 *(Esto se puede derivar del punto anterior, debido a que se tiene una reducción* LogSpace *de la expresión regular a un automata NFA libre de expresiones* ϵ*).*

Toda familia de autómatas que estudiamos son extensions de automata finito sobre cadenas por consiguiente el Lema 4.8 implica que todas las complejidades se dirigen a ser arriba de LogSpace. Este lema junto con el Lema 4.3 implica que la codificación del árbol de entrada es irrelevante, permitiendonos utilizar arbitrariamente la codificación de cadena o la codificación de apuntadores en función de que se requiera. Por otra parte la complejidad de $ArbolM(C)$ ahora dependerá de C.

Recordemos que una DTD está asociado a una expresión (única) regular para cada símbolo de Σ. Tenemos:

Teorema 4.9. *El ArbolM(DTD) es* NLogSpace*-completo bajo reducciones* uAC^0.

Prueba. La cota superior se demuestra como sigue. Usaremos un apuntador p_1 con el propósito de hacer el recorrido profundidad-primero del árbol de entrada. Otro apuntador p_2 se utilizará para particionar (to parse) a la DTD. Un apuntador extra p_3 sobre el árbol se usará para procesamientos locales. Para todo nodo apuntado por p_1 la palabra formada por las etiquetas de la lista de sus hijos se puede particionar en LogSpace utilizando p_3. El apuntador p_2 se posiciona a la expresión regular de la correspondiente DTD a la etiqueta apuntada por p_1. Es posible ahora verificar que la palabra apuntada por p_3 es de hecho en la expresión apuntada por p_2 en NLogSpace (ver Lema respectivo)

La cota inferior se deduce del hecho que el problema es obviamente más difícil que CadenaM(Expresión Regular) la cual es NLogSpace-completo por el Lema 4.8.

Para una DTD real, en el que se requieren expresiones regulares para ser no ambiguo , o para DTDs en el que se da un autómata determinista en lugar de una expresión regular la situación viene a ser más simple:

Teorema 4.10. *[Seg03] Tanto el ArbolM(DTD con DFA) como el ArbolM(XML DTD) son* LogSpace*-completo bajo reducciones* uAC^0.

Para el caso de un TWA la complejidad depende de que el autómata sea o no determinista. Lo siguiente extiende el resultado para un autómata de cadena de dos vías (two-way string automata):

Teorema 4.11. *Un ArbolM(DTWA) es* LogSpace*-completo bajo reducciones* uAC^0*, ArbolM(NTWA) esta en* NLogSpace*-completo bajo reducciones* uAC^0

Prueba: Asumimos que el árbol dado esta codificado en forma de una cadena. La cota inferior se obtiene directamente del Lema 4.8. Para la cota superior, la máquina de Turing necesita solamente dos apuntadores: uno para la posición actual de la cabeza sobre la cadena de entrada w, y otro apuntando al estado actual, ésto se puede hacer en LogSpace. Para obtener el siguiente paso, explora de manera determinista (o no determinista) la lista de transiciones del autómata hasta que obtenga una correspondencia al estado y a la etiqueta actual, y el movimiento del apuntador cabeza y del apuntador de estado esten en concordancia. Todo esto se puede hacer en LogSpace (o en NLogSpace respectivamente).

Para un autómata de árbol sobre un alfabeto con rango ArbolM se resuelve en [Loh01]:

- Tanto un *ArbolM(ConRango-NBUTA)* como un *ArbolM(ConRango-NTDTA)* son LogCFL-completo,
- Un *ArbolM(ConRango-DBUTA)* esta en LogDCFL mientras que un *ArbolM(ConRango-DTDTA)* está en LogSpace-completo.

Para *ArbolM(ConRango-NBUTA)*, la prueba de su dificultad en [Loh01] utiliza una reducción desde la membrecía a un lenguaje libre de contexto. Los datos del árbol por lo tanto corresponden burdamente al esqueleto de un árbol de derivación de la palabra de entrada y desde la CFG se construye un autómata que verifique que sin duda un árbol de derivación con una palabra de entrada se puede obtener a partir de los datos del árbol. La dificultad consiste en lograr esto dentro de un rango acotado ya que no hay razón *a priori* para que el árbol de derivación, con una palabra de entrada, tenga un rango acotado.

Venkateswaran [Ven91] establece una prueba del resultado con base a la caracterización uSAC^1 de LogCFL Como se ha hecho notar por varios autores, al verificar que un circuito uSAC^1 evalua a verdadero sobre una palabra de entrada w reduce al encontrar un *árbol prueba* para el circuito sobre w. Este *árbol prueba* se puede ver como un certificado que el circuito evalúa con resultado igual a 1. Esto hace evidente que, para un circuito en *forma normal*, el esqueleto de todos los *árboles de prueba* depende solamente de n y que puede computarse fácilmente en LogSpace. Ahora la reducción trabaja como sigue: dada una palabra de entrada w de tamaño n, el árbol de datos será el esqueleto de todos los *árboles de prueba* correspondientes a n, y el autómata de árbol será isomórfico al circuito C_n y de esta manera asigna un estado a cada nodo del árbol transformando el árbol de datos en un *árbol de prueba* si y sólo si el circuito evalúa a 1 teniendo a la entrada w.

Lema 4.9. *[Loh01] Tanto un ArbolM(ConRango-NBUTA) como un ArbolM(ConRango-NTDTA) están en* LogCFL*-completo.*

Prueba: La cota superior se obtiene de manera directa, al considerer la cota más baja. A continuación se muestra que la membrecía a un lenguaje uSAC^1 se puede reducir en LogSpace para

ArbolM(ConRango-NTDTA) o a

ArbolM(ConRango-NBUTA)

siguiendo un tratamiento similar. encontramos la definición de forma normal y de *árbol de prueba* para circuitos uSAC^1. Una familia uSAC^1 de circuitos lógicos está en la *forma normal* o NF (Normal Form) si las siguientes condiciones se satisfacen:

1. el fan-in de todas las compuertas AND es 2,
2. se puede asignar un nivel a los nodos de cada circuito de tal manera que las entradas están en nivel 0 y cada compuerta de nivel i recibe todas sus entradas desde los nodos de nivel $i-1$,
3. los circuitos están estrictamente alternados entre compuertas OR y compuertas AND (las compuertas de los niveles impares son OR y las compuertas de los niveles pares son AND),
4. todo circuito tiene un número impar de niveles, por consiguiente las compuertas de salida son AND.

Un *árbol de prueba* T para un circuito G de una familia uSAC en NF sobre una palabra de entrada w es un árbol con raíz junto con una función de etiquetado λ desde los nodos N de T hasta las compuertas de G de tal manera que:

1. la raíz de T esta etiquetada con la compuerta de salida G,
2. si $\lambda(N) = g$ y g es una compuerta AND entonces N tiene exactamente dos hijos N_1 y N_2 tales que $\lambda(N_1)$ y $\lambda(N_2)$ son las compuertas de entrada de g,
3. si $\lambda(N) = g$ y g es una compuerta OR entonces N tiene un hijo único N_1 tal que $\lambda(N_1)$ es una compuerta de entrada de g, y
4. cada hoja N de T esta etiquetada con una compuerta de entrada de G tal que el bit correspondiente es 1. Intuitivamente un *árbol de prueba* es una prueba que muestra que el circuito evidentemente evalúa a 1.

Podemos notar que para cada familia NF de uSAC^1, toda n y toda w de longitud n el árbol t del *árbol de prueba* T de G_n teniendo como entrada a w solamente depende de n (la función de etiquetado λ depende de w y de G_n).

A continuación nos dirigimos a la prueba del teorema. Sea $(G_n)_{n\in\mathbb{N}}$ una familia de circuitos NF en uSAC^1 y $w \in \{0,1\}^*$. Sea n la longitud de w. Tenemos que computar en LogSpace un árbol con rango t y un NTDTA con rango A tal que $t \in SAT(A)$ si y sólo si G_n acepta a w.

Sea t_n, el esqueleto del *árbol de prueba* para G_n con entrada w, donde a cada nodo se le asigna la misma etiqueta a. Sea:

$$A = (\{a\}, Q, q_0, \delta) \text{ donde } Q = V,$$

el conjunto de compuertas de G_n, en el que q_0 es la compuerta de salida de G_n y δ está definida como sigue:

- Si q es una compuerta AND con compuertas de entrada q_1 y q_2 entonces $\delta(a, q) = \{q_1 q_2\}$
- Si q es una compuerta OR entonces $\delta(a, q) = \{q' | q'$ es una compuerta de entrada $q\}$
- Si q es una compuerta de entrada de G_n entonces $\delta(a, q) = \{\epsilon\}$ si evalua a 1 de acuerdo con w.
- Si tenemos otro caso entonces $\delta(a, q) = \varnothing$.

Ahora estamos en posibilidad de verificar que $t \in SAT(A)$ si y sólo si t, junto con el etiquetado que se asigna a cada uno de sus nodos, el estado de A corresponde a un procesamiento de aceptación; forman un *árbol de prueba* de G_n con la entrada w si y sólo sí w es aceptada por G_n. La reducción, entonces esta en LogSpace ya que t_n puede computarse en LogSpace a partir de n y el cómputo en LogSpace de A se deduce a partir de la uniformidad LogSpace de los circuitos uSAC^1.

El resultado general para árboles sin rango fijo se obtiene por reducción al caso de rango fijo. Recordemos el Lema 4.5. Dado un lenguaje regular de árbol T el lenguaje de árbol $RANGO(T)$ es regular. A continuación sobrecargamos la notación de $RANGO$, y denotamos por $RANGO(A)$ a un autómata de árbol que reconoce al lenguaje regular de árbol $RANGO(SAT(A))$. Lo siguiente demuestra que la reducción se puede procesar en LogSpace.

Lema 4.10. *[GK+05]*

- *Dado un NTDTA A, el NTDTA $RANGO(A)$ se puede computar en* LogSpace.
- *Dado un NBUTA A, el NBUTA $RANGO(A)$ se puede computar en* LogSpace.

Prueba: Primero probamos el caso NTDTA dando la construcción LogSpace de $A' = RANGO(A)$. Sea $A = (\Sigma, Q, q_0, \delta)$ un NTDTA. Para cada letra $a \in \Sigma$ y cada estado $q \in Q$ sea $A^{a,q} = (Q, \Delta^{a,q}, p_0^{a,q}, F^{a,q}, \delta^{a,q})$ un autómata no determinista que reconoce el lenguaje regular de cadena $\delta(a, q)$, el alfabeto es Q y el conjunto de estados $\Delta^{a,q}$. Sea $\Delta = \bigcup_{a,q} \Delta^{a,q}$. Cada estado de A' sea una cuarteta de $\Sigma \times Q \times Q \times \Delta$ con el objetivo de grabar el lenguaje actual $A^{a,q}$ que se está procesando, la letra actual, y el estado actual en este lenguaje.

Sea $A' = (\Sigma^\sharp, Q', Q_0, \delta')$ el NTDTA definido por:

- $Q' = \{\langle a, q, q', p\rangle | a \in \Sigma, q \in Q, q' \in Q, p \in \Delta\} \cup q_\sharp$.
- $Q_0 = \{\langle a, q_0, q, p_0^{a,q_0}\rangle | a \in \Sigma, q \in Q\}$.
- $\langle a, q_i, q_j, p_0^{a,q_i}\rangle\langle b, q, q_k, p_k\rangle \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ si $\delta^{b,q}(q_i, p_i) = p_k$.
- $\langle a, q_i, q_j, p_0^{a,q_i}\rangle q_\sharp \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ si $p_i \in F^{b,q}$.

- $q_\sharp\langle b, q, q_k, p_k\rangle \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ si $\delta^{b,q}(q_i, p_i) = p_k$ y $p_0^{a,q_i} \in F^{a,q_i}$.
- $q_\sharp q_\sharp \in \delta'(a, \langle b, q, q_i, p_i\rangle)$ si $p_i \in F^{b,q}$ y $p_0^{a,q_i} \in F^{a,q_i}$.
- $\delta'(\sharp, q_\sharp) = \epsilon$.
- niguna otra cosa esta en δ'.

Se puede verificar que el cómputo anterior es correcto, A' corresponde a $RANGO(A)$ y esto puede realizarse dentro de LogSpace: se necesitan 4 apuntadores con el objetivo de definir un estado de la salida, por consiguiente son suficientes 12 apuntadores para calcular la función de transición δ'.

La prueba para NBUTA es similar. Sea $A = (\Sigma, Q, F, \delta)$ un NBUTA. Par cada letra $a \in \Sigma$ y cada estado $q \in Q$ sea:

$$A^{a,q} = (Q, \Delta^{a,q}, p_0^{a,q}, F^{a,q}, \delta^{a,q})$$

un autómata no determinista que reconoce el lenguaje regular de cadena $\{\overline{w}|\delta(a, w, q)\}$. Esto es computable en LogSpace partiendo de A. Sea Δ, $\bigcup_{a,q} \Delta^{a,q}$. Cada estado de A' será una tupla de $\Sigma \times Q \times \Delta$ para grabar el lenguaje actual que se está procesando y el estado actual en este lenguaje.

Sea $A' = (\Sigma^\sharp, Q', F', \delta')$ el NTDTA definido por:

- $Q' = \{\langle a, q, p\rangle | a \in \Sigma, q \in Q, p \in \Delta\}$
- $F' = \{\langle a, q, p\rangle | q \in F\}$.
- $\langle a', q', p''\rangle \in \delta'(b, \langle a, q, p\rangle\langle a', q', p'\rangle)$ si $a = b$, $p \in F^{a,q}$ y $\delta^{a',q'}(q, p') = p''$.
- $\langle a, q, p_0^{a,q}\rangle \in \delta'(\sharp, \epsilon)$.
- ninguna otra cosa esta en δ'.

Se puede verificar que el computo anterior es correcto, A' es evidentemente $RANGO(A)$ y esto se puede realizar dentro de LogSpace: Se requieren 3 apuntadores para definir un estado de la salida, por lo tanto son suficientes 9 apuntadores para computar la función de transición δ'.□

A partir de los lemas 4.10 y 4.9 se puede afirmar lo siguiente:

Teorema 4.12. *Arbol(NBUTA) y ArbolM(NTDTA) están en* LogCFL*-completo bajo reducciones* LogSpace*.*

5. CONSULTA DE DOCUMENTOS XML

El estudio de la complejidad al evaluar la consulta de documentos XML toma como base para este estudio a XPath (XML Path Lenguaje), recomendado por el W3C (16 de noviembre de 1999), para localizar información en un documento XML y navegar a través de sus elementos y atributos. La sintaxis de XPath define partes de un documento XML, utilizando expresiones de trayectoria (path expressions) que permiten realizar recorridos a través de ellos. Constituye el punto de inicio de otros lenguajes de consulta tales como XSLT y XQuery, esta diseñado para que sea utilizado por XSLT, XPointer y otras aplicaciones con software de particionado de archivos con formato XML.

5.1. Análisis del Lenguaje XPath

Las expresiones de trayectoria que utiliza XPath seleccionan nodos o conjuntos de nodos dentro de un documento XML, incluye alrededor de 100 funciones predefinidas para trabajar con valores de cadena, valores numéricos, comparaciones de fechas y tiempo, manipulación de nodos y de secuencias, así como valores boléanos.

En XPath se distinguen siete tipos de nodos: elemento (element), atributo (attribute), texto (text), espacio-con-nombre (namespace), instrucción de proceso (processing-instruction), comentario (comment), y raíz del documento (document-root). Si consideramos el siguiente documento XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
   <librería>
   <libro>
       <título idioma="ing">Elements of Finite Model Theory</título>
       <autor>L. Libkin</autor>
       <pub>2004</pub>
       <precio>29.99</precio>
   </libro>
</librería>
```

Podemos identificar los siguientes nodos: <librería>(nodo raíz del documento), <autor>L. Libkin </autor>(nodo elemento), idioma=“ing” (nodo atributo). Los valores atómicos (atomic values) son nodos sin hijos ni padre, por ejemplo: L. Libkin, o “ing”.

En XPath se definen las siguientes relaciones entre nodos:

Parent: Todo elemento y todo atributo tienen un padre; en el ejemplo, el elemento libro es el padre de título, autor, pub y precio.

Children: Todo nodo elemento puede tener ninguno, uno o más hijos; en el ejemplo título, autor, pub y precio son hijos del elemento libro.

Sibling: Es todo nodo con el mismo padre; en el ejemplo, título, autor, pub y precio son todos elementos que son hermanos.

Ancestor: Es todo nodo padre, o padre del padre, etc.; en el ejemplo los ancestros del elemento título son los elementos libro y librería.

Descendant: Es un nodo hijo, o hijo de hijo, etc.; en el ejemplo los descendientes del elemento librería son los elementos libro, título, autor, año y precio.

Las expresiones de trayectoria seleccionan nodos en un documento XML siguiendo un trayecto en forma de etapas, de la siguiente manera:

Expresión	Descripción
nodename	Selecciona todos los nodos hijos del nodo.
/	Selecciona a los nodos a partir del nodo raíz.
//	Selecciona los nodos a partir del nodo actual que concuerda con la selección sin importar donde se encuentren.
.	Selecciona el nodo actual.
..	Selecciona al padre del nodo actual.
@	Selecciona atributos.

Para los ejemplos a continuación presentados tomamos como base el siguiente documento XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<librería>
   <libro>
      <título idioma="ing"> Elements of Finite Model Theory </título>
      <precio>29.99</precio>
   </libro>
   <libro>
      <título idioma="ing">Learning XML</title>
      <precio>39.95</precio>
   </libro>
</librería>
```

Ejemplos de expresiones de trayectoria:

Expresión	Resultado
librería	Selecciona todos los nodos que son hijos del elemento librería.
/librería	Selecciona al elemento raíz de librería. Nota: Si la trayectoria empieza con una diagonal (/) representa en todos los casos una trayectoria absoluta para un elemento.
librería/libro	Selecciona todos los elementos libro que son hijos de librería.
//libro	Selecciona todos los elementos libro sin importar donde se encuentren en el documento.
librería//libro	Selecciona todos los elementos libro que son descendientes del elemento librería, sin importar que tan abajo de librería se encuentren.
//@idioma	Selecciona todos los atributos cuyo nombre es idioma.

Los predicados en XPath se utilizán para encontrar un nodo en particular o aquel que contiene un valor especifico, su notación incluye paréntisis cuadrados, como ejemplos podemos tener los siguientes:

Expresión XPath	Resultado
/librería/libro[1]	Selecciona el primer elemento libro que es hijo del elemento librería. Nota: en las versions recientes (2005) de los navegadores tienen implementada la convención que [0]representa el primer nodo, sin embargo el estándar del W3C establece que es [1].
/librería/libro[last()]	Selecciona el ultimo elemento libro que es hijo del elemento librería.
/librería/libro[last()-1]	Selecciona el último elemento libro pero que es el hijo del elemento librería.
/librería/libro[position()<3]	Selecciona a los dos primeroselementos libro que son hijos del elemento librería.
//título[@idioma]	Selecciona todos los elementos título que tienen un atributo con el nombre de idioma.
//título[@idioma=“ing”]	Selecciona todos los elementos título que tengan un atributo con nombre idioma y con valor de “ing”.
/librería/libro[precio>35.00]	Selecciona todos los elementos libro del elemento librería que tengan un elemento precio mayor a 35.00.
/librería/libro[precio>35.00]/título	Selecciona todos los elementos título de los elementos libro del elemento librería que tienen un elemento precio con un valor mayor a 35.00

Las expresiones XPath con comodines, a manera de las expresiones regulares, permiten seleccionar elementos XML desconocidos.

Comodin XPath	Descripción
*	Concuerda con cualquier nodo.
@*	Concuerda con cualquier atributo de un nodo.
node()	Concuerda con cualquier nodo de cualquier tipo.

Algunos ejemplos de expresiones de trayectoria con comodines se dan a continuación:

Expresión	Resultado
/librería/*	Selecciona todos los nodos hijo del elemento librería.
//*	Selecciona todos los elementos en el documento.
//título[@*]	Selecciona todos los elementos título que tengan cualquier atributo.

Con el operador | en una expresión de trayectoria XPath se pueden seleccionar varias trayectorias. A continuación se presentan algunos ejemplos:

Expresión	Resultado
//libro/título | //libro/precio	Selecciona todos los elementos título y (AND) elementos precio de todos los elementos libro.
//título | //precio	Selecciona todos los elementos título y (AND)todos los elementos precio en el documento.
/librería/libro/título | //precio	Selecciona todos los elementos título de libro que son a su vez elemento de librería y (AND) todos los elementos precio en el documento.

Un eje (Axe) en XPath define un conjunto de nodos relacionados con el nodo actual, como sigue:

Nombre del Eje	Resultado
ancestor	Selecciona todos los ancestros (padre, abuelo, etc.)del nodo actual.
ancestor-or-self	Selecciona a todos los ancestros (padre,abuelo, etc.) del nodo actual y se incluye el mismo en la selección.
attribute	Selecciona todos los atributos del nodo actual.
child	Selecciona todos los hijos del nodo actual.
descendant	Selecciona todos los descendientes (hijos, nietos, etc.) del nodo actual.
descendant-or-self	Selecciona todos los descendientes (hijos, nietos, etc.) del nodo actual y se incluye al mismo nodo en la selección.
following	Selecciona todo en el documento después de la marca de cierre del nodo actual.
following-sibling	Selecciona todos los hermanos después del nodo actual.
namespace	Selecciona todos los nodos en el espacio con nombre del nodo actual.
parent	Selecciona al padre del nodo actual.
preceding	Selecciona todo lo que esta en el documento antes de la marca de apertura del nodo actual.
preceding-sibling	Selecciona todos los hermanos antes del nodo actual.
self	Selecciona al nodo actual.

Las expresiones de Ubicación de Trayectoria (Location Path Expression) permiten localizar trayectorias de manera absoluta o relativa. Una trayectoria de ubicación absoluta inicia con una diagonal (/) y una relativa no la tiene, en ambos casos se compone de uno o mas etapas, cada una separada por una diagonal: una trayectoria absoluta: */step/step/...*, y una trayectoria relativa *step/step/...*. Cada uno de las etapas se evalúa contra los nodos en el conjunto del nodo actual, cada una de ellas se compone de:

- Un eje (axis) que define la relación triple entre los nodos seleccionados y el nodo actual)
- Una prueba de nodo (node-test) que identifica un nodo dentro del eje.
- Cero o más predicados (predicates) para una refinación más allá del conjunto de nodos (node-set) seleccionado.

La sintaxis para una etapa de localización es:

$$nombre - del - eje :: prueba - de - nodo[predicado]$$

Ejemplos:

Ejemplo	Resultado
child::libro	Selecciona todos los nodos libro que son hijos del nodo actual.
attribute::idioma	Selecciona el atributo idioma del nodo actual.
child::*	Selecciona todos los hijos del nodo actual.
attribute::*	Selecciona todos los atributos del nodo actual.
child::text()	Selecciona todos los nodos hijo de texto del nodo actual.
child::node()	Selecciona todos los nodos hijos de nodo actual.
descendant::libro	Selecciona todos los descendientes de libro del nodo actual.
ancestor::libro	Selecciona todos los ancestros de libro del nodo actual.
ancestor-or-self::libro	Selecciona todos los ancestros de libro del nodo actual incluyéndose el mismo en la selección si se trata de un nodo libro.
child::*/child::precio	Selecciona todos los nietos precio del nodo actual.

Con la finalidad de revisar en su conjunto la sintaxis básica de XPath tenemos los siguientes ejemplos, sobre el documento XML que a continuación presentamos (archivo "books.xml"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<librería>
   <libro categoria="MAT">
      <título idioma="esp">Algebra Lineal</título>
      <autor>Jaime Navarro</autor>
      <pub>2007</pub>
      <precio>30.00</precio>
   </libro>
   <libro categoria="LOG">
```

```
      <título idioma="ing">Elements of Finite Model Theory </título>
      <autor>L. Lubkin</autor>
      <pub>2005</pub>
      <precio>29.99</precio>
   </libro>
   <libro categoria="WEB">
      <título idioma="ing">XQuery Kick Start</título>
      <autor>James McGovern</autor>
      <autor>Per Bothner</autor>
      <autor>Kurt Cagle</autor>
      <autor>James Linn</autor>
      <autor>Vaidyanathan Nagarajan</autor>
      <pub>2003</pub>
      <precio>49.99</precio>
   </libro>
   <libro categoria="WEB">
      <título idioma="ing">Learning XML</título>
      <autor>Erik T. Ray</autor>
      <pub>2003</pub>
      <precio>39.95</precio>
   </libro>
</librería>
```

El archivo XML lo guardamos como "libros.xml", y considerando que lo consultamos a través de un navegador estándar, aplicamos el método implementation() para cargar el documento XML y el método evaluate() para seleccionar nodos dentro del documento, de esta manera tenemos:

```
xmlDoc=document.implementation.createDocument("","",null);
xmlDoc.async=false;
xmlDoc.load("libros.xml");
xmlDoc.evaluate(xpath, xmlDoc, null, XPathResult.ANY_TYPE,null);
```

Para seleccionar los títulos de todos los nodos libro, tenemos la siguiente sintaxis de trayectoria:

$$/librería/libro/título$$

La siguiente expresión de trayectoria selecciona solamente el título del primer nodo libro bajo el elemento librería:

$$/librería/libro[1]/título$$

La siguiente expresión de trayectoria selecciona el texto de todos los nodos precio:

$$/librería/libro/precio/text()$$

Para seleccionar los nodos precio con precio mayor a 35, podemos usar la siguiente expresión:

$$/librería/libro[precio > 35]/precio$$

Si con el mismo criterio anterior (precio>35) necesitamos seleccionar los títulos de los libros, lo podemos hacer con la expresión de trayectoria:

$$/librería/libro[precio > 35]/título$$

XPath es un lenguaje con un gran cantidad de prestaciones y por consiguiente demasiado amplio para un tratamiento teórico, en este análisis se introducen algunas restricciones y solamente tomamos algunas de sus propiedades claves. Corresponde de una manera general al Core XPath definido y utilizado en otros trabajos de investigación [GKP02] [GKP03].

El lenguaje Core XPath soporta las características de XPath usadas más comúnmente, conserva toda la potencia de navegación en trayectorias con las condiciones impuestas por los conectivos lógicos. No considera el valor de los datos por lo que carece de la posibilidad de concatenar cadenas y de realizar cómputos aritméticos sobre datos numéricos.

La sintaxis del Core XPath está definida por la gramática

```
locpath ::=    '/' locpath | locpath '/' locpath | locpath '|
               ' locpath | lockstep.
lockstep ::=   axis ': :' ntst '[' bexpr 'or' bexpr ']'.
bexpr ::=      bexpr 'and' bexpr | bexpr 'or' bexpr |
               'not('bexpr')' | locpath.
axis ::=       'self' | 'child' | 'parent' |
               'descendant' | 'descendant-or-self' |
               'ancestor' | 'ancestor-or-self' |
               'following' | 'following-sibling'
               'preceding' | 'preceding-sibling'.
```

"locpath" es la producción inicial, "axis" denota las relaciones eje, y "ntst" denota las marcas que etiquetan los nodos del documento o la estrella '*' que coincide con todas las marcas ("pruebas de nodo").

La principal característica sintáctica de Core XPath son las *trayectorias de ubicación* (location paths). A los predicados (predicates) se les denomina también *condiciones* (conditions).

La principal aplicación que conserva Core XPath es la navegación en árboles de documento XML. Esta se realiza utilizando las *relaciones eje* (axis relations), las relaciones binarias naturales como "hijo" (child) y "descendiente" (descendant) entre nodos, ya explicadas; también tienen los significados que conllevan sus nombres). El uso más común de XPath es el componer las aplicaciones eje con selección de nodos del documento por sus marcas ("pruebas de nodo"). Por ejemplo la expresión XPath:

$$/descendant :: a/child :: b$$

selecciona todos aquellos nodos etiquetados con b que son hijos de nodos etiquetados con a que son a su vez descendientes del nodo raíz (indicado por la diagonal inicial).

Las condiciones entre paréntisis cuadrados permiten aplicar restricciones adicionales en la selección de nodos. Por ejemplo,

$$/descendant :: a/child :: b[descendant :: candnot(following - sibling :: d)]$$

selecciona exactamente aquellos nodos v de entre los nodos obtenidos en el resultado de $/descendant :: a/child :: b$ que tengan *al menos un* descendiente etiquetado con c y que no tengan en el árbol un hermano derecho que este etiquetado con d).

Definición 5.1. *La sintaxis del "Fragmento Wadler" WF se define por la gramática del Core XPath con las siguientes extensiones: "bexpr" es ahora:*

bexpr ::=	*bexpr 'and' bexpr | bexpr 'or' bexpr | 'not'('bexpr')'|*
	locpath | nexpr relop nexpr.
expr ::=	*locpath | bexpr | nexpr.*
nexpr ::=	*'position()' | last()' | number | nexpr arithop nexpr.*
arithop::=	*'+' | '-' | '*' | 'div' | 'mod'.*
relop ::=	*'=' | '!=' | '<' | '<=' | '>' | '>='.*

"expr" (en lugar de "locpath") es ahora la producción inicial y "number" representa números constantes valuados como reales.

Ya que XPath se considera principalmente como un lenguaje para seleccionar un subconjunto los nodos de un árbol que representa el documento XML, nos podemos referir a las *expresiones* XPath como *consultas* (queries) XPath. Sin embargo notemos que los resultados de una consulta XPath también pueden ser de diferentes tipos, en particular para los numéricos y booleanos WF (así también como cadenas de caracteres si consideramos el XPath completo). Las expresiones XPath se evalúan con relación a un contexto, el cual por definición es una tupla de un *nodo-contexto* (context-node) y dos enteros, que se les llama *posición-contexto* (context-position) y el *tamaño-contexto* (context-size). Para más detalles nos podemos referir a la documentación del 3WC [2006], consideremos el siguiente ejemplo de consulta

$$child :: a[position() + 1 = last()].$$

Con respecto a la tripleta de contexto (v, i, j), i y j se ignoran cuando la etapa de ubicación child::a selecciona a los hijos de v que están etiquetados con "a". Sea $\{w_1, w_2, \ldots, w_m\}$ este conjunto de nodos, donde los índices corresponden al orden relativo de los nodos en el documento. La aplicación de un eje provoca un cambio de contexto al aplicar la condición [position()+1=last()]. La condición se pruebas para cada una de las tuplas $(w_1, 1, m) \ldots (w_m, m, m)$. Seleccionandose todos aquellos nodos w_k para los cuales $k + 1 = m$, esto es, seleccionará al nodo único $\{w_{m-1}\}$.

Como parte de la formalización, para analizar la complejidad de XPath, enunciamos el siguiente problema de decisión *Cons*:

[Cons] ENTRADA: Dada una consulta XPath Q, un árbol de datos t,
un contexto $\overline{c}$, y un valor r como resultado.
SALIDA: La evaluación de la consulta Q sobre el árbol de datos t,
para el contexto $\overline{c}$ da como resultado el valor r.

El resultado de una consulta XPath puede ser independiente de un contexto (por ejemplo las trayectorias de ubicación absolutas), en este caso tenemos la posibilidad de utilizar la notación corta $Q(t)$ para indicar el resultado de evaluar Q en el árbol de datos t.

Denotemos con $\mathcal{C}$ la complejidad de la clase y consideremos a X como un fragmento de XPath. Si el problema de decisión CONS, que evalúa consultas XPath a partir del fragmento X, está en la clase de complejidad $\mathcal{C}$ o en $\mathcal{C} - dificil$ o bién en $\mathcal{C} - completo$, entonces podemos afirmar que el fragmento X esta en $\mathcal{C}$, en $\mathcal{C} - dificil$, o en $\mathcal{C} - completo$ respectivamente.

El problema de evaluar una consulta XPath está en PTime [GKP02] con respecto a la complejidad combinada.

Las consultas Core XPath se puden evaluar en tiempo $\mathcal{O}(|Q| \cdot |t|)$ [GKP02] en donde $|Q|$ representa el tamaño del árbol de datos.

Una consulta para CoreXPath puede evaluarse en tiempo $\mathcal{O}(n, q)$ donde n es el tamaño del documento XML y q es el tamaño de la consulta (query) a Core XPath. De una manera más formal utilizaremos la siguiente gramática de expresiones XPath:

exp	::=	path | /path
path	::=	step $(/step)^*$
step	::=	sort :: Σ|$sort :: \Sigma[pred]$
pred	::=	pred $\cup$ pred | pred $\cup$ pred | $\neg$ pred | exp

donde *sort* es un elemento de

{ *parent (padre), ancestor (ancestro), ancestor-or-self (ancestro-o-el mismo), preceding (precedente), preceding-sibling (hermano-precedente), self (el mismo), child (hijo), descendant (descendiente), descendant-or-self (descendiente-o-el-mismo), following (siguiente), following-sibling (hermano-siguiente)* }.

Una expresión XPath regresa un conjunto de nodos que satisfacen sus restricciones de trayectoria. Nos podemos referir a [XPa08]para una semántica más precisa. Intuitivamente, $e[e']$, cuando evalúa un nodo p, regresa los nodos n tales que la trayectoria de p hacia n verifican e de tal manera que existe un nodo m para el cual la trayectoria desde n hasta m satisface e'. El significado de cada *etapa* es intuitivo.

Dado un árbol etiquetado t y una expresión XPath expresión e, denotamos por $e(t)$ al conjunto de nodos que satisfacen las restricciones de la trayectoria de e. De nuevo dividimos el análisis en dos partes. La primera tratará a la *complejidad de los datos* al evaluar expresiones XPath en la que la consulta es fija y solamente los datos son variables. La segunda parte considera su *complejidad combinada* en la que la consulta es parte de la entrada.

5.2. Complejidad de los Datos

Sea e una expresión de Core XPath. Consideremos el siguiente problema de decisión:

[Eval(e)] ENTRADA: un árbol t y un nodo n
SALIDA: verdadero si y sólo sí $n \in e(t)$.

De nuevo la codificación del árbol de entrada es crucial para la complejidad de los datos. Consideremos las dos codificaciones anteriormente presentadas la tipo DOM y la tipo SAX.

Teorema 5.1. *[GKP03] Sea e una expresión XPath, Si t es una cadena, Eval(e) esta en* uTC^0. *Si t es una estructura de apuntadores, Eval(e) esta en* LogSpace .

Prueba: Consideremos primero el caso en el que t esta codificada como cadena. Mostramos que Eval(e) se puede expresar mediante una fórmula FOM formula φ_e(root,y) tal que t, i, j $\models \varphi_e(x, y)$ si y sólo sí el nodo apuntado por j en t satisfacen $e(t)$ cuando inicia la evaluación en el nodo i. Entonces este se prueba a partir de dos antecedentes ya analizados:

- $\mathsf{FOM} = \mathsf{uTC}^0$
- Si t esta en forma de estructura de cadena, la codificación en forma de apuntador se puede procesar en uTC^0 a partir de $[t]$.

Esto se hace por inducción sobre e. Por ejemplo, si e es $e'/step :: a$, entonces:

$$\varphi_e(x, y) \text{ es } \exists z \varphi_{e'}(x, z) \wedge step(z, y) \wedge Q_a(y),$$

donde $step(x, y)$ es la fórmula FOM que verifica que el nodo x y y están relacionados como lo describe *step*. De la misma tenemos que si e es $e_1[e_2]$, entonces $\varphi_e(x, y)$ es:

$$\varphi_{e_1}(x, y) \wedge \exists z \varphi_{e_2}(x, y)$$

Cuando t esta dado en forma de estructura de apuntadores, la cota superior se obtiene directamente de la complejidad ya revisada en el capítulo anterior:

1. Si t esta en forma de estructura de apuntadores, $[t]$ se puede procesar en LogSpace a partir de $[t]$.
2. Si t esta en forma de estructura de cadena, la codificación en forma de apuntador se puede procesar en uTC^0 a partir de $[t]$

□

Las cotas superiores planteadas por este teorema son próximas. Recordemos que de hecho en el análisis del problema ORD, en el capítulo anterior, se verifica que en una rama, el nodo etiquetado a ocurre antes de un nodo etiquetado b. El cómputo de ORD se ejecuta en LogSpace-completo bajo reducciones uAC^0. El resultado se obtiene ya que esta propiedad se puede expresar en XPath.

Es conveniente mencionar que el XPath completo tiene la habilidad de contar el número de nodos que satisfacen una expresión dada y realiza pruebas sobre el resultado. Debido a que FOM tiene también esta capacidad, extendiendo el anterior fragmento del núcleo y con los hechos planteados, si consideramos que la aritmética permanece dentro de uTC^0 (en uTC^0 se pueden realizar comparaciones, sumas y multiplicaciones), no afecta a los resultados del teorema anterior.

5.3. Complejidad Combinada

Para la complejidad combinada, e es parte de la entrada, si suponemos que el árbol esta dado como una estructura de apuntadores, formalmente el problema que estudiaremos se plantea como sigue:

[Eval(XPath)] ENTRADA: an XPath expresión e,
un árbol t y un nodo n de t
SALIDA: verdadero ssi $n \in e(t)$.

En los siguientes algoritmos, la consulta XPath se ve como un árbol en el que cada nodo está etiquetado ya sea por una operación booleana ($\vee$, $\wedge$, $\neg$) o bién por una expresión de etapa (step-expression) de la forma $pred :: a$ donde $a \in \Sigma$. Se construye el árbol t_e, correspondiendo a una expresión del núcleo de XPath e, de manera inductiva como sigue. Si $e = e_1/e_2$ donde e_1 es una expresión de etapa y e_2 una expresión de trayectoria (path-expression), entoces t_e se construye sumando un nodo (raíz) nuevo de etiqueta e_1 a t_{e_2} con una arista a la raíz de t_{e_2}. Si $e = e_1[e_2]/e_3$ en la que e_1 es una expresión de etapa, e_3 una expresión de trayectoria y e_2 una expresión pred (pred-expression), entonces t_e se construye sumando dos nuevos nodos n y m hacia t_{e_2} y t_{e_3}. Sus etiquetas son respectivamente e_1 y $\vee$. n es la raíz de t_e y tiene una arista hacia m. m tiene una arista para cada una de las raíces de t_{e_2} y t_{e_3}. Si $e = e_1 \cup e_2$ entonces t_e se construye a partir de t_{e_1} y t_{e_2} al agregar un nuevo nodo raíz n con etiqueta $\vee$ y conectado a las raices de t_{e_1} y t_{e_2}. Los otros operadores booleanos se tratan de manera similar. Una marca especial † se agrega para la hoja del árbol correspondiente a la última expresión de etapa de la expresión de trayectoria que no aparece como un predicado. Por ejemplo el árbol asociado a la expresión XPath

$$next :: a[descendent :: b \cup descendent :: c]/next :: d$$

se puede representar mediante el siguiente diagrama:

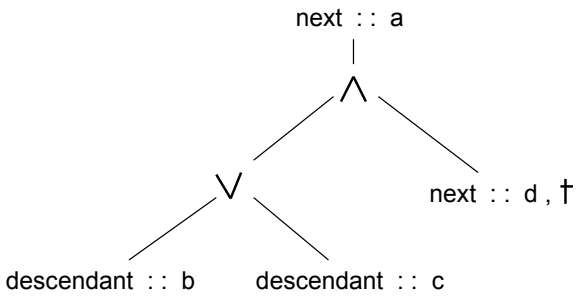


Fig. 5.1: Árbol asociado a una expresión

Este árbol de consulta nunca se materializará, sin embargo todas las consultas se particionarán (parsed) al realizar una travesía depth-first (primero-profundidad) left-first (primero-izquierda) del árbol correspondiente. Un apuntador hacia una expresión XPath siempre señalará al primer caracter de una expresión de paso, o a un operador booleano, por consiguiente corresponderá a un nodo del árbol de consulta. Notemos que, dada una expresión XPath como una cadena, el apuntador al nodo próximo en esta travesía depth-first (primero-profundidad) se puede calcular en LOGSPACE a partir de la posición actual.

Consideraremos inicialmente a las expresiones Core XPath que no contienen la negación, es decir sea $Eval(XPath-)$ el caso especial de $Eval(XPath)$ en el que las expresiones XPath no contienen el conectivo negación. En este caso la evaluación se puede lograr conjeturando los datos del nodo que corresponde a cada nodo del árbol de consulta, utilizando una pila para almacenar los cómputos parciales.

Teorema 5.2. *[GKP03] $Eval(XPath-)$ esta en* LogCFL.

Prueba: Dado un apuntador i en una consulta e su *profundidad* se define como la cantidad de nodos del tipo expresiones de paso que ocurren en la trayectoria desde el nodo n (correspondiente a i) hasta la raíz del árbol correspondiente a e, incluyendo n. En el ejemplo anterior, la profundidad del nodo etiquetado "*descendent* :: c" es 2.

Utilizando la caracterización LogCFL de Sudborough [Sud78] y considerando un autómata de pila (pushdown) auxiliar no determinista que reconoce a $Eval(XPath)$ en espacio logarítmico y tiempo polinómico. El autómata mantiene un apuntador p hacia la expresión XPath de entrada, que realizará una travesía en preorden (preorder traversal) del árbol de consulta. Como hemos visto, esta travesía en preorden se puede lleva a cabo en LogSpace. El autómata utilizará otro apuntador q, sobre el árbol de entrada, que se dirige al nodo del árbol de entrada que hará verdadera a la consulta en p. El autómata inicia posicionando a p y q hacia las raíces correspondientes a la consulta de entrada y al árbol de entrada, respectivamente, operando como sigue.

Sea p' el sucesor de p en el recorrido en preorden de la consulta, y n el nodo apuntado por q (si no se tiene sucesor de p el autómata regresa y acepta). Supongamos primero que p' es un descendiente de p. El autómata tiene que verificar que la expresión de trayectoria correspondiente es verdadera. Si la etiqueta de p' es booleana, p' es saltada y el autómata se mueve al siguiente nodo. Si la etiqueta de p' es una expresión de etapa, el autómata propone un nodo n' del árbol de entrada de tal forma que la trayectoria desde n hasta n' satisfaga a la expresión de etapa (si tal nodo no existe entonces el automata rechaza). Si p' contiene la marca especial † entonces el autómata verifica que n' corresponde al nodo de entrada, de otra manera lo rechaza. Entonces empuja (push) al apuntador de n' en la pila, reposiciona (reset) p y q a sus nuevos valores p' y q' y procede con el bucle principal (main loop) con estos nuevos valores.

Ahora consideramos que p' es un *uncle* (tío) de p. Sea d la profundidad actual de p y p'' el hermano de p' quien es un ancestro de p. En este punto el autómata conoce que el subárbol con raíz en $p"'$ evalúa a verdadero. Entonces el autómata posiciona a p al primer ancestro de p' cuya etiqueta no es un $\vee$. Sea d' la nueva profundidad de p, el autómata extrae (pops) de la

pila en un tiempo $d - d'$ y posiciona a q al valor en la cima de la pila. A continuación procede como lo hizó anteriormente con estos valores para p, p', q, q'.

Se puede comprobar que el algoritmo es correcto ya que acepta si y sólo si $n \in e(t)$. Utiliza solamente un apuntador hacia su entrada y por consiguiente se procesa en LogSpace. Para cada nodo en el árbol de consulta, exactamente hace una conjetura y una verificación para una expresión de paso. La última verificación se puede hacer en un tiempo $\mathcal{O}(|t|)$, por lo tanto el autómata trabaja en un tiempo $\mathcal{O}(|e| \cdot |t|)$. Esto prueba el teorema. □

Notemos en primer lugar que la variante determinista directa del algoritmo anterior trabaja en tiempo exponencial con la máxima profundidad del árbol de consulta de la expresión XPath, por consiguiente no es evidente todavía si $Eval(XPath-)$ está o no en LogDCFL.

En [Str94] se proponen técnicas de conteo que muestran la cerradura de LogCFL bajo complemento, en una primera instancia, podemos vernos tentados a utilizar esta técnica con el propósito de generalizar el teorema anterior, con expresiones XPath que utilizan la negación y el conteo. Desafortunadamente, como se muestra también en la misma referencia [Str94] , cada una de las operaciones de conteo requiere un tiempo polinomial en función del tamaño de los datos. Por lo tanto el tiempo de proceso del PDA auxiliar no determinista será exponencial en función de la negación anidada (nested negation) y el conteo, y como consecuencia nos lleva a una clase superior a LogCFL (al menos que acotemos el anidamiento). Se sabe que el PDA auxiliar no determinista procesa en espacio logarítmico y en tiempo exponencial caracterizado por PTIME.

El teorema anterior muestra que $XPath-$ es paralelizable y se puede realizar con recursos limitados de memoria como LogCFL $\subseteq$ DSPACE($\log^2$). Al evaluar XPath con recursos limitados es importante (para secuencias continuas o streaming XML por ejemplo, ver [SeV02] , a continuación se plantean dos casos especiales no triviales de $Eval(XPath)$ que pueden evaluarse en LogSpace. Sea $Eval(XPath^1)$ la restricción de $Eval(XPath)$ en la que cada expresión de paso tiene un ordenamiento en:

{ previous-sibling, self, child, parent, next-sibling};

o bien que $Eval(XPath^{+*})$ en la que cada expresión de paso tiene un ordenamiento en:

{ self, descendant, descendant-self};

. $Xpath^1$ permite solamente expresiones de paso que pueden ir a un nodo con distancia cuando mucho de 1 desde el nodo actual, mientras que $Xpath^{+*}$ permite solamente etapas con grandes distancias en el futuro.

Tenemos:

Teorema 5.3. *[GKP03]* $Eval(XPath^1)$ *es* LogSpace.$Eval(XPath^{+*})$ *es* LogSpace*-completo.*

Prueba. En el caso de $XPath^1$ el primer paso del algoritmo elimina las expresiones de etapa que observa *backward: parent* y *previous sibling*. Esto se puede realizar en LogSpace aplicando

un algoritmo más general que elimine todas las expresiones de paso *backward* en cualquier expresión XPath. El algoritmo requiere en general tiempo exponencial pero se puede verificar que el mismo algoritmo, cuando se restringe a expresiones de $XPath^1$ esta en LogSpace. Por lo tanto podemos considerar que las expresiones $XPath^1$ solamente contienen expresiones de etapa *forward*.

En ambos casos el algoritmo realiza una travesía primero-en-profundidad del árbol de la expresión XPath, y para cada nodo particiona los datos utilizando un recorrido primero-en-profundidad primero-a-la-izquierda con el propósito de encontrar un nodo que coincida. En cada etapa, únicamente almacena el apuntador actual hacia el árbol y el apuntador hacia el nodo coincidente en el árbol de datos. Esto requiere un espacio logarítmico en función del tamaño de la entrada. Utiliza para los cómputos locales espacio logarítmico extra . El proceso consiste en que estando en coincidencia, el recorrido hacia atrás (backtracking) se puede realizar en LogSpace. En el caso de $XPath^1$, esto se realiza hacia adelante (straightforward), ya que cada etapa se puede ejecutar en orden inverso. En el caso de $Xpath^{+*}$, se debe al hecho que, para cada rama, es suficiente considerar la primer posible coincidencia, ya que la recursión $*$ toma a su cargo todas las demás.

Recordemos que las expresiones booleanas se pueden validar en LogSpace saltándose las disyunciones tan pronto como uno de sus subárboles evaluén a 1 y sucesivamente evaluando todos los subárboles de una conjunción. Las negaciones se manejan de manera similar. Con el propósito de simplificar la presentación, a continuación ignoraremos los nodos del árbol etiquetado con predicado booleano y siempre evaluaremos todos sus subárboles. Utilizando la evaluación de la expresión booleana descrita anteriormente estamos en posibilidad, de manera directa, a extenderlo para incluir compuertas lógicas mientras mantenemos las restricciones LogSpace.

El algoritmo funciona de manera similar en ambos casos. Sea next una subrutina que compute el recorrido primero-en-profundidad del árbol de la expresión XPath. Esto se puede realizar en LogSpace. Los casos de $Eval(XPath^1)$ y $Eval(XPath^{+*})$ solamente difieren en las subrutinas backtrack y fetch.

fetch(β, i, j) toma como entrada dos apuntadores i y j para los datos del árbol y un apuntador β para el árbol de consulta. Regresa un apuntador k hacia un nodo tal que, k sigue estrictamente a j en el orden primero-en-profundidad primero-a-la-izquierda en el árbol de datos, y la trayectoria de i a k satisface el predicado de paso de la etiqueta de β. Si tal nodo no existe, regresa el valor de falso. Esta subrutina, en ambos casos, se puede llevar a cabo en LogSpace.

backtrack(α, i, β) toma como entrada dos apuntadores α y β para el árbol de consulta, de tal manera que el padre δ de β es un ancestro de α, y un apuntador i para el árbol de datos. Entonces computa un apuntador hacia un nodo de datos k tal que: la trayectoria desde k hasta i satisface a la expresión de trayectoria del subconjunto de la consulta que va desde δ hasta α. En el caso de $XPath^1$ esta k es única y se puede computar en LogSpace. En el caso de $Xpath^{+*}$ regresamos el ancestro de i que esta más cerca de la raíz, de tal manera que la trayectoria desde la raíz hasta k satisfase las restricciones de trayectoria dadas por la consulta de la subconsulta dada por la trayectoria desde la raíz hasta δ. Se puede mostrar que esta k se computa en LogSpace y que esta estrategia funciona con el siguiente algoritmo:

/* initialization */
α = root(query tree)
i = root(data tree)
if label(α) $\neq$ label(i) return FALSE
/* main loop */
1 if done(α) return TRUE
/* process the next node of the query */
β= next(α)
/* i was the data node corresponding to α */
$i = backtrack(\alpha, i, \beta)$
/* it is now the data node corresponding
to the father of β */
α = father(β)
$j = i$
/* find a matching data node in
a depth-first way */
2 if ($fetch(\beta, i, j)$) /* there is a matching j*/
then $i = j$;$\alpha = \beta$; goto (1)
else /* backtrack */
if $i = root$ return FALSE
else $i = backtrack(\alpha, i\alpha)$
/* i is now the data node corresponding
to the father of α */
$\beta = \alpha$; $\alpha = father(\beta)$; $j = i$
goto (2)

Con estos antecedentes, las cotas inferiores para $Eval(XPath^{+*})$ se comprueban por una reducción a ORD. □

5.4. XPath y Lógicas Temporales

5.4.1. Lógica CTL para Árboles sin Rango Fijo

.

Como hemos visto, los formalismos lógicos para modelar los documentos XML se basan en la MSO, para lo referente a su verificación, y en la FO, en lo que respecta a su consulta. Existe una conexión entre estas dos lógicas y las lógicas temporales, cuando se analizan los árboles con rango (ranked trees), es decir aquellos árboles cuyos nodos tienen una cantidad fija de hijos. Por ejemplo CTL es equivalente a FO sobre estructuras arborescentes con una satisfactibilidad en EXPTIME. La conexión entre XPath y CTL ha sido ampliamente estudiada, en particular entre las lógicas LTL y CTL.

El *dominio D de un árbol sin rango fijo* es un conjunto finito de prefijos cerrado formado con cadenas de números naturales, tales que si $s \cdot i \in D$ entonces $s \cdot j \in D$, para todo $j < i$. Si Σ

es un alfabeto finito, entonces un Σ-arbol sin rango fijo es una estructura de primer orden

$$T = \langle D \prec_{ch}, \prec_{sb}, \prec^*_{ch}, \prec^*_{sb}, (P_a)_{a\in\Sigma}\rangle$$

donde $\prec_{ch}$ se interpreta como la relación *child* (hijo):

$$(s \prec_{ch} s\cdot i \quad \text{si} \quad s, s\cdot i \in D)$$

$\prec_{sb}$ como el orden de los hermanos (sibling):

$$(s\cdot i \prec_{sb} s\cdot(i+1) \quad \text{si} \quad s\cdot(i+1), s\cdot i \in D)$$

Ya que los documentos XML se modelan como árboles sin rango fijo con hermanos ordenados, se pueden ver como sistemas de transición etiquetados, más aún muchas de las tareas que involucra la navegación mediante trayectorias en un documento recuerda las propiedades temporales de las trayectorias en los sistemas de transición. En términos de expresividad, las lógicas próximas a la consulta XML son la FO y MSO. Pero desde el punto de vista eficiencia de la evaluación de la consulta, no son las mejores

Redefinimos la sintaxis de las lógicas tipo LTL sobre un alfabeto Σ como:

$$\varphi, \varphi' := a, a \in \Sigma | \varphi \vee \varphi' | \neg\varphi | X\varphi | X^-\varphi | \varphi U \varphi' | \varphi S \varphi'$$

Las fórmulas de LTL, se interpretan sobre cadenas finitas o infinitas formadas a partir de un alfabeto Σ, una fórmula se evalúa con base a una posición en una cadena. Dada una cadena $s = a_0 a_1 \ldots$, la semántica se define como sigue:

- $(s,i) \models a$ ṫ si y sólo si $a_i = a$;
- $(s,i) \models X_\varphi$ (next φ) si y sólo si $(s, i+1) \models \varphi$;
- $(s,i) \models X^-\varphi$ si y sólo si $(s, i-1) \models \varphi$;
- $(s,i) \models \varphi U \varphi'$ (φ until φ' si existe $j \geq i$ tal que $(s,j) \models \varphi'$ y $(s,k) \models \varphi$ para todo $i \leq k < j$;
- la semantica de la dual $\varphi S \varphi'$ (φ since φ') es que existe $j \leq i$ tal que $(s,j) \models \varphi'$ y $(s,k) \models \varphi$ para todo $j < k \leq i$.

A continuación consideremos la *lógica temporal en árbol* $\mathsf{TL}^{\mathsf{tree}}$ definida como sigue:

$$\varphi, \varphi' := a, a \in \Sigma | \varphi \vee \varphi' | \neg\varphi | X_*\varphi | X^-_*\varphi | \varphi U_* \varphi' | \varphi S_* \varphi'$$

en donde $*$ corresponde ya sea a *ch* (child) o *ns* (next sibling), definimos a continuación su semántica respecto a un árbol y a un nodo en un árbol:

- $(T,s) \models a$ si y sólo si $\lambda_T(s) = a$;

- $(T, s) \models X_{ch}\varphi$ si y sólo si $(T, s\dot{i}) \models \varphi$ para alguna i;
- $(T, s) \models X^{-}_{ch}\varphi$ si y sólo si $(T, s') \models \varphi$ para el nodo s' tal que $s' \prec_{ch} s$;
- $(T, s) \models \varphi U_{ch}\varphi'$ si existe um nodo s' tal que $s \prec^{*}_{ch} s', (T, s') \models \varphi'$, y para toda $s'' \neq s'$ que satisfase $s \prec^{*}_{ch} s'' \prec^{*}_{ch} s'$ tenemos $(T, s'') \models \varphi$.

La semántica de S_{ch} se define invirtiendo el orden de la semántica de U_{ch}, y la semántica de X_{ns}, X^{-}_{ns}, U_{ns}, y S_{ns} es la misma remplanzando la relación *child* (hijo) con la relación *next sibling* (siguiente hermano).

TL^{tree} define naturalmente las consultas unarias sobre árboles, y también define las consultas booleanas: decimos que $T \models \varphi$ si $(T, \epsilon) \models \varphi$.

Teorema 5.4. *[Mar04] Una consulta unaria o booleana sobre árboles sin rango fijo es definable en FO sí y solo si es definible en* TL^{tree}.

En las lógicas del tipo CTL^{*}, tenemos dos clases de fórmulas: aquellas evaluadas en nodos de un árbol, y las que son evaluadas sobre las trayectorias de un árbol. Esto es similar a la situación con XPath, lo cual tiene expresiones filtro evaluadas en los nodos, y expresiones de ubicación en trayectoria, las cuales se evalúan en trayectorias de árboles XML. A continuación definimos dos lógicas: CTL^{*} que incluye el pasado, y la reformulación de XPath a través de una lógica tipo CTL como se presenta en [Mar04] .

El lenguaje XPath condicional [Mar04] o CXPath (Condicional XPath), se define para que contenga *formulas de nodo* α y *formulas de trayectoria* (path formulae) β, como:

$$\begin{array}{lll} \alpha, \alpha' & := & a, a \in \Sigma | \neg\alpha | \alpha \vee \alpha' | E\beta \\ \beta, \beta' & := & ?\alpha | step | (step/?\alpha)^{+} | \beta/\beta' | \beta \vee \beta' \end{array}$$

en donde *step* (paso) es uno de los siguientes:

$$\prec_{ch}, \prec^{-}_{ch}, \prec_{ns} \quad o \quad \prec^{-}_{ns}$$

$E\beta$ establece la existencia de una trayectoria , y / corresponde a la composición secuencial de trayectorias.

Formalmente, si tenemos un árbol T, se evalúa en cada nodo una fórmula nodal (esto es, definimos $(T, s) \models \alpha$), y cada fórmula de trayectoria que relaciona dos nodos (esto es, $(T, s, s') \models \beta$). La semántica corresponde a lo siguiente:

- $(T, s) \models a$ si y sólo si $\lambda_T(s) = a$;
- $(T, s) \models E\beta$ si y sólo si existe s' tal que $(T, s, s') \models \beta$;
- $(T, s, s') \models ?\alpha$ si y sólo si $s = s'$ y $(T, s) \models \alpha$;
- $(T, s, s') \models step$ si y sólo si $(s, s')' \in step$;
- $(T, s, s') \models \beta/\beta'$ si y sólo si para alguna s'' tenemos que $(T, s, s'') \models \beta'$ y $(T, s'', s') \models \beta'$;

- $(T, s, s') \models (step/?\alpha)^+$ si existe una secuencia de nodos:

$$s = s_0, s_1, \ldots, s_k = s',$$

para $k > 0$, tal que cada (s_i, s_{i+1}) es un paso, y $(T, s_{i+1}) \models \alpha$ para todo $i < k$.

El lenguaje Core XPath se obtiene permitiendo $step^+$ como opuesto a $(step/?\alpha)^+$ en la definición de las fórmulas de trayectoria. Notemos que debido a que $step^+ = (step/?verdadero)$, donde:

$$verdadero = \bigvee_{a \in \Sigma} a$$

tenemos que: $CoreXPath \subseteq CXPath$.

Si tenemos la expresión XPath:

$$(\quad //a \quad [\quad //b \quad] \quad /c \quad)$$

se puede representar en esta sintaxis por la fórmula nodal:

$$c \vee E(\prec_{ch}^- / \quad ?a/ \quad \prec_{ch}^+ / \quad ?b)$$

lo cual nos dice que un nodo esta etiquetado con c , y existe una trayectoria que empieza con dirección a su padre, donde encuentra a a, y a continuación se dirige a un descendiente de esa a y encuentra una b.

Core XPath corresponde a XPath tal como lo define el 3WC, mientras que CXPath representa una adición a XPath propuesto en [Mar04]. Esta adición consiste esencialmente en el operador *until* de la lógica temporal: por ejemplo, para representar la versión estricta de until (es decir, que en el próximo elemento de una trayectoria se cumple con aUb), podemos expresar esto como:

$$\prec_{ch} / \quad ?b \quad \vee \quad (\prec_{ch} / \quad ?a)^+ / \quad ?b.$$

Las formulas nodales tanto de CXPath como de Core XPath definen consultas unarias sobre árboles, esto se puede caracterizar como sigue:

Teorema 5.5. *[Mar04]*

- *El poder de las fórmulas nodales de CXPath corresponde precisamente al poder de las consultas unarias de* FO.
- *Las fórmulas nodales de Core XPath tiene precisamente el poder de las consultas unarias* FO^2 *(es decir* FO *con dos variables) en el vocabulario:*

$$\prec_{ch}, \quad \prec_{ch}^*, \quad \prec_{ns}, \quad \prec_{ns}^*$$

Las lógicas CTL y CTL^* corresponden a lógicas temporales con ramificación que se utilizan ampliamente en la verificación de sistemas reactivos, y están definidas sin conectivos hacia el pasado. En estas lógicas también se consideran tanto fórmulas en los nodos (denominadas

estados) como fórmulas de trayectoria, pero estás últimas se evalúan solamente en trayectorias y no de manera arbitraria sobre pares de nodos.

Definimos una fórmula de nodo α en CTL^*_{past}, así como fórmulas de trayectorias a un hijo β_{ch} y a un hermano β_{ns} como sigue:

$$\begin{array}{rcl} \alpha, \alpha' & := & a(a \in \Sigma)|\neg\alpha|\alpha \vee \alpha'|E\beta_{ch}|E\beta_{ns} \\ \beta_{ch}, \beta'_{ch} & := & \alpha|\neg\beta_{ch}|\beta_{ch} \vee \beta'_{ch}|X_{ch}\beta_{ch}|X^-_{ch}\beta_{ch}|\beta_{ch}U_{ch}\beta'_{ch}|\beta_{ch}S_{ch}\beta'_{ch} \\ \beta_{ns}, \beta'_{ns} & := & \alpha|\neg\beta_{ns}|\beta_{ns} \vee \beta'_{ns}|X_{ch}\beta_{ns}|X^-_{ch}\beta_{ns}|\beta_{ns}U_{ns}\beta'_{ns}|\beta_{ns}S_{ns}\beta'_{ns} \end{array}$$

Si tenemos un árbol, una trayectoria a un hijo (child-path) π_{ch} es una secuencia de nodos sobre una trayectoria desde la raíz hasta una hoja, y una trayectoria a un hermano (sibling-path) es una secuencia π_{ns} de nodos de la forma:

$$s \cdot 0, \ldots, s \cdot (n-1)$$

para un nodo s con n hijos, definimos la semántica de una fórmula nodal con respecto a un nodo en un árbol, y de una fórmula de trayectoria con respecto a una trayectoria, asi como a un nodo sobre la trayectoria (es decir, definimos la noción de $(T, \pi_*, s) \models \beta_*$, en la que $*$ puede corresponder a ch o a ns)

- $(T, s) \models E\beta_*$ si existe una trayectoria π_* tal que $s \in \pi_*$ y $(T, \pi_*, s) \models \beta_*$;
- $(T, \pi_{ch}, s) \models X_{ch}\beta$ si $(T, \pi_{ch}, s') \models \beta$, donde s' es el hijo de s en la trayectoria π_{ch};
- $(T, \pi_{ch}, s) \models X^-_{ch}\beta$ si $(T, \pi_{ch}, s') \models \beta$, donde s' es el padre de s en π_{ch};
- $(T, \pi_{ch}, s) \models \beta U_{ch}\beta'$ si para alguna $s' \neq s$ tal que $s' \in \pi_{ch}$ y $s \prec^*_{ch} s'$, tenemos que $(T, \pi_{ch}, s') \models \beta'$, y para todo $s \prec^*_{ch} s'' \prec^*_{ch} s', s'' \neq s'$, tenemos que $(T, \pi_{ch}, s'') \models \beta$.

La definición de S_{ch} y la correspondiente a las trayectorias hacia un hermano (sibling-phaths) son análogas.

Lo siguiente se puede ver como una analogía de la equivalencia $\mathsf{FO} = \mathsf{CTL}^*$ para árboles binarios finitos. Mientras que la prueba de la conexión entre un árbol con rango a uno sin rango fijo, la traducción directa a partir de un árbol binario falla debido a que las trayectorias sobre las traducciones de árboles sin rango fijo, pueden cambiar de dirección entre las trayectorias hacia un hijo y hacia un hermano, haciendo esto de manera arbitraria y de forma repetitiva.

Teorema 5.6. *[BaL05] Una consulta unaria o booleana sobre arboles sin rango fijo es definible en* FO *si y sólo si es definible en* CTL^*_{past}.

5.4.2. Consulta XML vía Verificación de Modelos con CTL

.

La opción de realizar la consulta de documentos XML a través de la verificación de modelos (model checking) se puede materializar [Afa2004], debido a que uno de los principales temas de

esta diciplina algorítmica es el de encontrar una representación eficiente de modelos susceptibles de ser verificados con esta disciplina. Sobre esta línea de investigación se han desarrollado varias técnicas, en particular *la verificación simbólica de modelos*, las cuales han demostrado, en el caso de la consulta XML, su factibilidad teórica planteada por Libkin [Lib05] y su implementación, utilizando el verificador NuSMV, por Franceschet y Afanasiev [Afa2004]. Obteniendo las cotas de complejidad lineal que nos permite la aplicación de la lógica CTL,.

Los algoritmos evaluados corresponden a dos fragmentos del XPath, el lenguaje ya revisado en este estudio, Core XPath, y el CXPath, propuesto por Marx [Mar04], que constituye una variante más expresiva que el Core XPath, que llega a constituir un lenguaje expresivamente completo respecto a la FO sobre árboles XML. La sintaxis de CXPath es como sigue:

Definición 5.2. *(Lenguaje CXPath) Sea Σ un conjunto de etiquetas que corresponde a las marcas de los elementos XML. Una consulta CXPath es una fórmula generada por la primera cláusula de la siguiente definición inductiva.*

$$\begin{array}{rcl} locationpath & ::= & /locationstep|locationstep|locationpath/locationstep \\ locationstep & ::= & axis :: l|axis :: l[pred] \\ pred & ::= & pred\ and\ pred|pred\ or\ pred|\ not\ pred|locationpath \\ axis & ::= & \epsilon|d|d^*|(d[pred])^*|([pred]d)^{*|} \end{array}$$

donde $l \in \Sigma \cup \{*\}$ y $d \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$.

CXPath es más expresivo que Core XPath [Mar04], Core XPath esta contenido en CXPath, debido a que los 11 ejes de XPath se pueden definir en terminos de CXPath, como se indica a continuación:

$$\begin{array}{rcll} self & :: l[pred] & \equiv & \epsilon :: l[pred] \\ child & :: l[pred] & \equiv & \downarrow :: l[pred] \\ parent & :: l[pred] & \equiv & \uparrow :: l[pred] \\ descendant & :: l[pred] & \equiv & \downarrow :: */ \downarrow^* :: l[pred] \\ ancestor & :: l[pred] & \equiv & \uparrow l[pred] :: */ \uparrow^* :: l[pred] \\ descendant\ or\ self & :: l[pred] & \equiv & \downarrow^* :: l[pred] \\ ancestor\ or\ self & :: l[pred] & \equiv & \uparrow^* :: l[pred] \\ following\ sibling & :: l[pred] & \equiv & \rightarrow :: */ \rightarrow^* :: l[pred] \\ preceding\ sibling & :: l[pred] & \equiv & \leftarrow :: */ \leftarrow^* :: l[pred] \\ following & :: l[pred] & \equiv & \uparrow^* :: */ \rightarrow :: */ \rightarrow^* :: */ \downarrow^* :: l[pred] \\ preceding & :: l[pred] & \equiv & \uparrow^* :: */ \leftarrow :: */ \leftarrow^* :: */ \downarrow^* :: l[pred] \end{array}$$

Para cualquier relación binaria R, denotemos su inverso como R^{-1}, es decir:

$$R^{-1} = \{ \quad (m, n) \quad | \quad (n, m) \in R \quad \}.$$

, y a R^* como la cerradura transitiva y reflexiva, es decir la relación transitiva y reflexiva más pequeña que contiene R.

La semántica de una consulta CXPath sobre un árbol XML T esta dada por las siguientes tres funciones definidas inductivamente simultaneas:

- $\{\{\cdot\}\}_T$ que, dada una consulta CXPath, regresa la correspondiente relación binaria sobre el conjunto de nodos de T,

- $[[\cdot]]_T$ que dado un predicado, regresa un conjunto de nodos de T que satisfacen el predicado,
- $\langle\langle\cdot\rangle\rangle_T$ que, dado un eje, regresa la relación binaria correspondiente sobre el conjunto de nodos de T.

Definición 5.3. *(Semántica de CXPath [Mar04]) Sea $T = (N, R_\downarrow, R_\rightarrow, L)$ un árbol XML. $R_\uparrow = (R_\downarrow)^{-1}$, $R_\leftarrow = (R_\rightarrow)^{-1}$ y sea $L(*) = N$. Sea $root \in N$ sea la raíz del árbol T. Se define la semántica de CXPath como sigue:*

$$
\begin{aligned}
\{\{axis :: l\}\}_T &= \{(n,m) \in \langle\langle axis\rangle\rangle_T | m \in L(l)\} \\
\{\{axis :: l[pred]\}\}_T &= \{(n,m) \in \langle\langle axis\rangle\rangle_T | m \in L(l) \ y \ m \in [[pred]]_T\} \\
\{\{/locationstep\}\}_T &= \{(n,m) | n \in N \ y \ (root, m) \in \\
&\quad \{\{locationstep\}\}_T\} \\
\{\{locationpath/locationstep\}\}_T &= \{(n,k) | \exists m.(n,m) \in \{\{locationpath\}\}_T \ y \\
&\quad (m,k) \in \{\{locationstep\}\}_T\} \\
[[pred_1 \ and \ pred_2]]_T &= [[pred_1]]_T \cap [[pred_2]]_T \\
[[pred_1 \ or \ pred_2]]_T &= [[pred_1]]_T \cup [[pred_2]]_T \\
[[notpre]]_T &= N \backslash [[pred]]_T \\
[[locationpath]]_T &= \{n | \exists m.(n,m) \in \{\{locationpath\}\}_T\} \\
\langle\langle\epsilon\rangle\rangle_T &= \{(n,n) | n \in N\} \\
\langle\langle d\rangle\rangle_T &= R_d \\
\langle\langle d^*\rangle\rangle_T &= R_d^* \\
\langle\langle (d[pred])^*\rangle\rangle_T &= \{(n,m) \in R_d | m \in [[pred]]_T\}^* \\
\langle\langle ([pred]d)^*\rangle\rangle_T &= \{(n,m) \in R_d | n \in [[pred]]_T\}^*
\end{aligned}
$$

El resultado de la evaluación de una consulta CXPath q sobre un árbol T es el conjunto:

$$Result(T, q) = \{m | \exists n.(n,m) \in \{\{q\}\}_T\}$$

Podemos observar que, dada una consulta $q = /q'$, el resultado de la consulta absoluta q y la correspondiente a q' pueden diferir.

Con estos antecedentes, estamos en condiciones de traducir los modelos y consultas para CXPath a modelos y formulas CTL. Reducimos la evaluación de la consulta CXPath al modelo de verificación CTL. Como primer paso, convertimos un modelo de árbol XML a un modelo CTL.

Definición 5.4. *(Árboles XML) Sea Σ el conjunto de las etiquetas correspondientes a las marcas XML. Un árbol XML es un árbol con nodos etiquetados y hermanos ordenados:*

$$T = (N, R_\downarrow, R_\rightarrow, L),$$

donde N es el conjunto de nodos, $R_\downarrow \subseteq N \times N$ es el conjunto de las aristas del árbol, $R_\rightarrow \subseteq N \times N$ es una relación funcional que asocia cada nodo con su hermano inmediato ubicado a la derecha (si lo tiene), y $L : \Sigma \to \mathcal{P}(N)$ es una función de etiquetado de nodo que asocia a cada etiqueta un conjunto de nodos.

Definición 5.5. *(De árboles XML a modelos $\mathsf{CTL}_{\{\uparrow,\downarrow,\leftarrow,\rightarrow\}}$) Sea un árbol XML:*

$$T = (N, R_{\downarrow}, R_{\rightarrow}, L) \text{ con } L : \Sigma \to \mathcal{P}(N)$$

Definimos la traducción de T a una gráfica etiquetada:

$$\mu_1(T) = (N, (E_d)_{d\in\{\uparrow,\downarrow,\leftarrow,\rightarrow\}}, L),$$

con: $E_{\downarrow}$, $E_{\rightarrow}$, $E_{\uparrow} = (R_{\downarrow})^{-1}$, $E_{\leftarrow} = (R_{\rightarrow})^{-1}$

El modelo que obtenemos $\mu_1(T)$ corresponde en la mayoría de los casos total con respecto a:

$$\bigcup_{d\in\{\uparrow,\downarrow,\leftarrow,\rightarrow\}} E_d$$

lo cual nos permite hablar de trayectorias en el modelo, la única excepción es el modelo $\mu_1(T)$, cuando T es un árbol XML con un sólo nodo, en este análisis no tomamos en consideración este caso.

Podemos notar que el tamaño de $\mu_1(T)$ es linear respecto al tamaño de T, y que $\mu_1(T)$ se puede obtener a partir de T en tiempo lineal.

Dado un eje *axis*, sea $axis^{-1}$ su inverso, es decir:

$$\epsilon^{-1} = \epsilon, (\leftarrow)^{-1} = \rightarrow, \ldots$$

y $(d^*)^{-1} = (d^{-1})^*$. Y para $l \in \Sigma \cup \{*\}$, sea:

$$l' = \begin{cases} \text{true} & \text{para } l = *, \\ l & \text{para otro caso.} \end{cases}$$

Definición 5.6. *(De consultas CXPath a formulas $\mathsf{CTL}_{\{\uparrow,\downarrow,\leftarrow,\rightarrow\}}$) Dada una consulta CXPath q, definimos su traducción a una fórmula CTL $\tau_1(q)$ como se muestra a continuación, donde*

$$root = \neg EX_{\uparrow} true.$$

$$\begin{aligned}
\tau_1(axis :: l) &= l' \wedge \langle axis^{-1}\rangle true \\
\tau_1(axis :: l[pred]) &= l' \wedge \langle axis^{-1}\rangle true \wedge \omega_1(pred) \\
\tau_1(/axis :: l) &= l' \wedge \langle axis^{-1}\rangle root \\
\tau_1(/axis :: l[pred]) &= l' \wedge \langle axis^{-1}\rangle root \wedge \omega_1(pred) \\
\tau_1(locationpath/axis :: l) &= l' \wedge \langle axis^{-1}\rangle \tau_1(locationpath) \\
\tau_1(locationpath/axis :: l[pred]) &= l' \wedge \langle axis^{-1}\rangle \tau_1(locationpath) \wedge \omega_1(pred) \\
\omega_1(pred_1 \text{ and } pred_2) &= \omega_1(pred_1) \wedge \omega_1(pred_2) \\
\omega_1(pred_1 \text{ or } pred_2) &= \omega_1(pred_1) \vee \omega_1(pred_2) \\
\omega_1(notpred) &= \neg\omega_1(pred)
\end{aligned}$$

$$\begin{array}{rcl} \omega_1(axis :: l) & = & \langle axis \rangle l' \\ \omega_1(axis :: l[pred]) & = & \langle axis \rangle (l' \wedge \omega_1(pred) \end{array}$$

$$\begin{array}{rcl} \langle \epsilon \rangle \alpha & = & \alpha \\ \langle d \rangle \alpha & = & EX_d \alpha \\ \langle d^* \rangle \alpha & = & EU_d(true, \alpha) = EF_d \alpha \\ \langle (d[pred])^* \alpha & = & EU'_d(\omega_1(pred), \alpha) \\ \langle ([pred]d)^* \alpha & = & EU_d(\omega_1(pred), \alpha) \end{array}$$

Se puede observar que en la traducción los casos en los que se considera un predicado, a pesar que no siguen una definición recursiva de una trayectoria de ubicación, son exhaustivos y deterministas. También se puede notar que la longitud de $\tau_1(q)$ puede obtenerse a partir de q en tiempo lineal.

Así mismo, podemos ver al eje *axis* en un paso de ubicación dentro de una consulta como una modalidad CTL *observando en dirección opuesta*, mientras que el eje en un paso de ubicación para un predicado, se puede ver como una modalidad CTL *en la misma dirección*. La razón subyacente consiste en que la traducción CTL de un predicado caracteriza los nodos que permanecen en el *inicio* de esa trayectoria. De esta manera la longitud de una consulta no es otra cosa que la profundidad de la fórmula CTL correspondiente.

Por ejemplo si consideramos la consulta absoluta XPath q:

```
q = /child::libro [child::autor [following_sibling::autor]]/child::título
```

Su equivalente, en la notación que estamos utilizando, corresponde a:

$$q = / \downarrow :: libro[\downarrow :: autor[\rightarrow :: */ \rightarrow_* :: autor]]/ \downarrow :: título$$

Selecciona los nodos *título* de todos los *libros* con al menos dos *autores*.

Su traducción $\tau_1(q)$ corresponderá a la siguiente fórmula CTL:

$$título \wedge EX_{\uparrow}(libro \wedge EX_{\uparrow} root \wedge EX_{\downarrow}(autor \wedge EX_{\rightarrow}(true \wedge EX_{\rightarrow} autor))$$

la cual es equivalente a:

$$título \wedge EX_{\uparrow}(libro \wedge EX_{\uparrow} root \wedge EX_{\downarrow}(autor \wedge EX_{\rightarrow} EF_{\rightarrow} autor))$$

6. CONCLUSIONES

Como elemento central de la globalización del intercambio informático a través de la Web, ahora enfilada a su era semántica, XML se consolida como su protagonista. Apuntalado tanto por sus aspectos metalingüísticos, normativos y prácticos, al igual que por los resultados teóricos que formalizan sus bondades, salvando el amplio espectro de análisis a la que han sido sometidos los algoritmos que le dan operatividad, incluyendo la "prueba del ácido" que la complejidad computacional permite aplicar; como reacción equivalente a su relevancia presente y su importancia potencial.

Esta diversa abundancia analítica de la *intelligenza* informática, reúne sobre un mismo sujeto, un rico conjunto de herramientas teóricas, que permiten dar seguimiento a una metodología de modelado formalizado, que combina principios de la lógica, los autómatas y la complejidad computacionales, que de manera paralela permiten abordar los niveles que le siguen a la pieza "leggo"que representa XML, en la construcción de la estructura que se construye en el marco de la Web Semántica, como la organización de información en un *Marco de Descripción de Recursos o RDF (Resource Description Framework)*, y la utilización de un *Lenguaje para Ontologías en la Web o OWL (Web Ontology Language)*.

Los procesos sobre documentos XML estudiados son la validación mediante el sistema DTD, y la consulta a través de un fragmento de expresiones XPath denominado *Core XPath*. Con el objetivo de evaluar la eficiencia de tales algoritmos se ubican las cotas teóricas superiores de complejidad. La importancia de analizar el lenguaje radica en que nos habilitó para revisar en detalle y al límite asintótico, estos dos procesos esenciales: Validar y recorrer un documento XML.

La validación, como se ha explicado, distingue los documentos XML que tienen sentido de aquellos que no lo poseen. Un caso común se presenta cuando dos partes quieren intercambiar alguna información utilizando documentos XML; requieren, en términos de la estructura, un acuerdo acerca del formato de los documentos que aceptan, por ejemplo un documento con raíz en *libros* tiene uno o más hijos con etiquetas *libro*, cada una de las cuales deben tener cuatro hijos con etiquetas (ordenadas) *titulo*, *autor*, *fecha*, y *editor*, y un hijo opcional con etiqueta *agotado*. La clase de formalismos que hemos requerido para este proceso corresponde a un autómata de árbol (TA) de anchura no definida no determinista (NUTA).

La consulta de un documento XML implica su recorrido, por ejemplo podemos preguntar por todos los nodos, dentro del documento, que están marcados con *libro* y tiene un hijo etiquetado como *agotado*. Se ha visto que para esta navegación, la lógica que rige es la FO, esto se debe a que sus fragmentos están estrechamente conectados con expresiones en XPath, un lenguaje, que en la práctica, como hemos revisado, se utiliza para hacer recorridos en una estructura

jerárquica tipo árbol.

El documento XML se modela mediante un LUOT, lo cual es una práctica común en la literatura consultada (ver por ejemplo [Nev02] y [Suc01]), ya que permite formalizar las especificaciones de tipificación, los lenguajes de consulta utilizan precisamente un autómata de árbol sin rango fijo (UTA); este elemento ha posibilitado la aplicación de los principios acerca de estos autómatas de árbol (TA) reunidos sistemáticamente y formalizados en el libro de difusión abierta y libre reunida bajo el título de Tree Automata Techniques and Applications (TATA) [CD+07], ampliamente recomendado por el Instituto Nacional de la Investigación en Ciencia Informática y en Control (INRIA) de Francia. Los TA con alfabeto fijo y estructura con rango definido (ranked) han sido profusa y fructíferamente estudiados motivados por sus aplicaciones. La generalización de estos autómatas hacia árboles sin rango fijo se revisó en el contexto de los algoritmos XML.

El tratamiento preliminar con que se enfrenta un documento XML, es el de su codificación, para esta fase se consideraron los dos modelos correspondientes a los utilizados por las aplicaciones recomendadas por el 3WC: el primero es una estructura de apuntadores, inspirada en la usada en el modelo SAX, la otra entrega una cadena con la sucesión de las marcas de apertura y cierre encontradas durante el recorrido primero-en-profundidad (depth-first) de la estructura arborescente del documento, como el utilizado en el modelo de datos DOM.

Siguiendo la metodología de análisis presentado por Libkin [Lib04] para modelos finitos, se consideran dos variantes en la presentación del problema de la verificación XML. En la primera, el tipo es fijo y la entrada consiste solamente en el árbol de datos (complejidad de los datos), mide la complejidad en función exclusivamente del tamaño de los datos, proporcionando una buena aproximación al comportamiento del problema cuando el tamaño del tipo se presume que no es importante comparado al tamaño de los datos. En la segunda, el tipo es variable ya que también, junto con los datos, es parte de la entrada (complejidad combinada), su propósito se dirige a medir con mayor precisión la dificultad del problema

El primer proceso estudiado correspondió a la verificación, modelado mediante un árbol etiquetado y considerando los sistemas de tipificación DTD y DTD extendido, su representación como un TA y su generalización hacia sus diferentes modalidades (top-down, bottom-up, determinista, no determinista y walking). Consideramos dos codificaciones que reflejan los dos modelos más utilizados para XML: el primero es una estructura de apuntador como la usada en el modelo DOM, la segunda es una cadena que corresponde a la sucesión de las marcas de apertura y cierre encontradas durante el recorrido primero en profundidad (depth-first) del árbol, como el utilizado en el modelo de datos SAX. En el exámen de los algoritmos se muestra, como se señala en [GKP03], que en el problema de la validación XML, la complejidad de los datos depende fundamentalmente de la codificación del árbol de entrada, ya que la clase de complejidad involucrada corresponde a la franja inferior de la jerarquía (por debajo de LogSpace). Mientras que la pertenencia de un árbol a un lenguaje definido por un TA con alfabeto de anchura definida, estudiada en [Loh01], no puede ampliarse, de manera inmediata, a árboles con anchura o rango variable porque la codificación utilizada para árboles con anchura definida (una cadena basada en una notación prefija del árbol) ya no es válida para árboles sin rango fijo.

Para la codificación tipo DOM se muestra que, con base en [GKP03], para todos los sistemas de tipificación aquí considerados, la complejidad del problema de validación esta en LogSpace. Para la codificación tipo SAX encontramos que, para todos los sistemas de tipificación considerados, el problema de la complejidad esta en uNC^1. Esto por consiguiente amplia los resultados para árboles sin rango definido con codificación tipo SAX. La complejidad combinada del problema de validación depende de la sintaxis del sistema de tipificado, ubicándose en una complejidad que va desde LogSpace-completo hasta LogCFL-completo.

Para un NDTA sobre alfabetos con anchura definida se muestra que esta en LogCFL-completo utilizando una reducción no trivial de la membresía a un lenguaje libre de contexto (CFL). El estudio, en esta parte de la revisión, confirma que la extensión no es directa como en el caso de la codificación de un autómata en árboles sin anchura definida, es más compleja que para árboles en un alfabeto de anchura definida, ya que cada transición corresponde a una expresión regular en lugar de un conjunto finito de palabras. Para un DUTA se obtienen niveles de complejidad ligeramente superiores que para el caso de tener una anchura definida.

Otro resultado de interés que se revisa referente a la transformación que requerimos de un árbol con rango no fijo a uno con rango definido, consiste en que su codificación se puede ejecutar en uTC^0 si esta en formato SAX, y en LogSpace cuando es tipo DOM, así como que el árbol resultante se procesa en LogSpace.

En cuanto a la revisión de la complejidad teórica de la consulta de documentos XML, considerando el fragmento Core XPath, se muestra que puede evaluarse en tiempo polinomial, y que la complejidad de los datos esta en uTC^0.

Se ha examinado que la verificación de la pertenencia de un ULT es un lenguaje de árbol regular no es más difícil que la verificación de la pertenencia de una cadena en un lenguaje regular de cadena, en consonancia con esto la complejidad de los datos de la verificación de la conformidad de un documento XML es independiente del sistema de tipificado aplicado. Así mismo se confirma que la codificación utilizada para el árbol de datos es determinante y que los dos modelos considerados nos arrojan cotas de complejidad diferentes: uno uNC^1 y el otro LogSpace.

En el trabajo se analizó la complejidad combinada para la validación XML, observandose que ésta no depende de la codificación del árbol de entrada pero si del lenguaje de tipificación que se utilice, también se deduce que los niveles de complejidad del proceso varían desde LogSpace-completo hasta LogCFL-completo. En cuanto a la complejidad combinada para la consulta XML, utilizando Core XPath, se confirma, como se establece en [GKP03], que corresponde realmente a P-completo, así mismo se ratifica que el algoritmo para la evaluación simple de una consulta se ubica en LogCFL-completo.

Por último el estudio muestra las evidencias que la verificación de modelos con CTL ofrece una ópción importante, para la evaluación de consultas de documentos XML, con cotas de complejidad lineal, aportando las numerosas y maduras herramientas de verificación, que esta dinámica y popular diciplina a logrado concitar en su desarrollo.

Bibliografía

[ABS99] S. Abiteboul, P. Buneman, y D. Suciu, *Data on the Web: from relations to semistructured data and XML*, 1999.

[Afa2004] L. Afanasiev, *XML Query Evaluation via CTL Model Checking*, ILLC master thesis with advisoring of M. Franceschet, 2004.

[ALR00] E. Allender, M. Loui, y K. Regan, *Complexity Classes*, National Science Foundation, 2000.

[BaC89] D. A. M. Barrington, y J. Corbett, *On the relative complexity of some languages in NC1*, Information Processing Letter, 32:251-256, 1989.

[BaL05] P. Barcelo, y L. Libkin, *Temporal Logics over Unranked Trees*, In Proc. IEEE Symp. on Logic in Comp. Sci., pages 31-40, 2005

[BeM99] C. Beeri, y T. Milo. *Schemas for integration and translation of structured and semistructured data*. In Proc. of Intl. Conf. on Database Theory, 1999.

[Ber96] L. Bertossi, *Lógica para Ciencia de la Computación*, Universidad Católica de Chile, 1996.

[BIS90] D. A. M. Barrington, N. Immerman, y H. Straubing. *On Uniformity within NC1*. Journal of Computer and System Sciences, 41(3):274306, 1990.

[Bru98] A. Bruggemann-Klein, y D. Wood, *One-Unambiguous Regular Languages*. Information and Computation, 142(2), 1998.

[Cai03] J. Cai, *Lectures in Computational Complexity*, University of Winsconsin, 2003.

[CCG03] C. Calcagno, L. Cardelli, y A. Gordon, *Deciding Validity in a Spatial Logic for Trees*, TLDI03, 2003.

[CD+07] H. Comon, M. Dauchet, R. Gilleron, F. Jacqemard, D. Lugiez, C. Löding , S. Tison, and M. Tommasi, *Tree Automata Techniques and Applications*, disponible en `http://www.grappa.univ-lille3.fr/tata`, 2007.

[DOM08] Document Object Model (DOM), Disponible en `http://www.w3c.org/dom`.

[FoS02] L. Fortnow, S. Homer, *A Short History of Computational Complexity*, Boston University, 2002.

[GaJ79] M. Garey, y D. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[GKP02] G. Gottlob, C. Koch, y R. Pichler. *Efficient Algorithms for Processing XPath Queries*. In Proc. of Intl. Conf. on Very Large Data Bases, 2002.

[GKP03] G. Gottlob, C. Koch, y R. Pichler. *The Complexity of XPath Query Evaluation*. In Proc. ACM Symp. on Principles of Database Systems, 2003.

[GK+05] G. Gottlob, C. Koch, R. Pichler, y L. Segufin. *The Complexity of XPath Query Evaluation and XML Typing*. Journal of the ACM, Vol. 52, No. 2, pp 284-335, 2005.

[HMU00] J. Hopcroft, R. Motwani, y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 2000.

[HuR00] M. Huth, y M. Ryan, *Logic in Computer Science. Modelling and reasoning about systems*, Cambridge University Press, 2000

[Imm99] N. Immerman, *Descriptive Complexity*, Springer-Verlag, 1999.

[Jul05] P. Julian I., *Logica Simbolica*, Alfaomega, 2005.

[LaR96] R. Lassaigne, y M. de Rougemont, *Logique et Complexite*, Hermes, 1996.

[Lib04] L. Libkin, *Elements of Finite Model Theory*, Springer-Verlag, 2004.

[Lib05] L. Libkin, *Logics over Unranked Trees: An Overview*, 32nd ICALP, 2005.

[Loh01] M. Lohrey, *On the Parallel Complexity of Tree Automata*, In Rewriting Techniques and Applications (RTAŠ01), 2001.

[McN67] R. McNaughton, *Parenthesis Grammars*, Journal of Computer and System Sciences, 1967.

[Mar04] M. Marx, *Condicional XPath, the first order complete XPath dialect*, In ACM Symp. on Principles of Database Systems, pp.178-187, 2004

[Nev02] F. Neven. *Automata, Logic, and XML*, In Proc. of Computer Science Logic, 2002.

[NWB00] A. Navaro, W. White, y L. Burman, *Mastering XML*, SYBEX, 2000.

[Pap94] C. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.

[PaV03] Y. Papakonstantinou, y V. Vianu, *Incremental Validation of XML Documents*. In Proc. of Intl. Conf. on Database Theory, 2003.

[PeM05] E. Pérez, y R. Mac Kinney, *Análisis de Algoritmos*, Universidad Autónoma Metropolitana, 2005.

[Pen98] R. Pena M., *Diseño de Programas, Formalismo y Abstracción*, Prentice Hall, 1998.

[SAX08] Simple API for XML (SAX). Disponible en `http://www.saxproject.org/`.

[Seg03] L. Segoufin, *Typing and Querying XML Documents:Some Complexity Bounds*, PODS, 2003.

[SeV02] L. Segoufin, y V. Vianu, *Validating streaming XML documents*. In Proc. ACM Symp. on Principles of Database Systems, 2002.

[Str94] H. Straubing, *Finite Automata, Formal Logic, and Circuit Complexity*, Birhauser, 1994.

[Suc01] D. Suciu, *Typechecking for Semistructured Data*. In Proc. workshop on Database Programming Language, 2001.

[Sud78] I. Sudborough, *On the Tape Complexity of Deterministic Context-Free Languages*, Journal of ACM, 25(3), pp. 405-414, 1978

[Ven91] H. Venkateswaran, *Properties that characterize LOGCFL*. Journal of Computer and System Sciences, 43:380-404,1991.

[Via01] V.A. Vianu, *Web Odyseey: from Codd to XML*, In Proc. 20th Symposium on Principles of Databasr Systems (PODS 2001), pp. 1-15, 2001.

[W3C08] *The World Wide Web Consortium (W3C)*, `http:/www.w3.org/` . Los creadores de los estándares XML, DOM, SAX, DTD y muchos otros relacionados; en esta página se encuentran las versiones más recientes de los estándares así como información adicional y enlaces a otras fuentes de información, 2008.

[Wid04] A. Widjaja, *The Limits of Computation*, Melbourne University, 2004.

[XCP08] The XML Cover Pages, `http:/www.oasis-open.org/cover/sgml-xml.html.` Un servidor del grupo OASIS.

[XML08] xml.com, `http:/www.xml.com` . La página de O'Reily dedicada al XML.

[XPa08] XML Path Language (XPath). Disponible en `http://www.w3.org/TR/2002/WD-xpath20-20020816.`

[YeP87] Y. Yershov, y Y. Paliutin, *Lógica Matemática*, Nauka, 1987.