ACERCANDO EL CONOCIMIENTO EN LA WEB A LOS AGENTES SOFTWARE

UN MARCO DE TRABAJO PARA LA CONSTRUCCIÓN DE WRAPPERS SEMÁNTICOS

JOSÉ LUIS ARJONA

UNIVERSIDAD DE SEVILLA

TESIS DOCTORAL

DICIEMBRE DE 2004

Support: Trabajo financiado parcialmente por el Miniterio de Ciencia y Tecnología (proyectos TIC-2000-1106-C02-01, FIT-150100-2001-78 y TIC-2003-02737-C02-01), y por la Junta de Castilla la Mancha (proyecto PCB-02-001).

005 173

17-01-05

Don Rafael Corchuelo Gil, Profesor Titular de Universidad del Área de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla,

HACE CONSTAR

que don José Luis Arjona Fernández, Ingeniero en Informática por la Universidad de Sevilla, ha realizado bajo mi supervisión el trabajo de investigación titulado

Acercando el Conocimiento en la Web a los Agentes Software.
Un Marco de Trabajo para la Construcción de Wrappers Semánticos

Una vez revisado, autorizo el comienzo de los trámites para su presentación como Tesis Doctoral al tribunal que ha de juzgarlo.

Fdo. Rafael Corchuelo Gil
Profesor Titular de Universidad
Área de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Sevilla, diciembre de 2004

UNIVERSIDAD DE SEVILLA

COMISIÓN DE DOCTORADO

UNIVERSIDAD DE SEVILLA
REGISTRO GENERAL
TERCER CICLO
SALIDA
Nº. 200500200001436
18/03/2005 13:22:54

Sevilla, 16 de Marzo de 2005
N/Ref.: Negociado de Tesis EL/CAR
Asunto: Enviando Tesis Doctoral Leída

ILMO. SR. DIRECTOR DE LA BIBLIOTECA
DE LA E.T.S. DE INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE SEVILLA

Adjunto le remito ejemplares de Tesis Doctorales leídas en Departamentos vinculados a esa Facultad a fin de que pasen a formar parte de fondos bibliográficos de consulta de ese Centro.

AUTORES DE LAS TESIS LEÍDAS

- ARJONA FERNÁNDEZ, JOSÉ LUIS

LA JEFA DE SECCIÓN DE DOCTORADO

UNIVERSIDAD DE SEVILLA

Fdo.: Yolanda Díaz Rolando.

José Luis Arjona Fernández, con DNI número 44.218.635-P,

DECLARA

Ser el autor de la tesis que se presenta en esta memoria, y cuyo nombre es:

Acercando el Conocimiento en la Web a los Agentes Software.
Un Marco de Trabajo para la Construcción de Wrappers Semánticos

Lo cual firmo en Sevilla, diciembre de 2004.

Fdo. José Luis Arjona Fernández

A Sina
Por dar significado a mi vida.

Índice general

Índice de figuras

Índice de cuadros

Agradecimientos

No cabe duda que uno de los mejores momentos en el desarrollo de una tesis doctoral es aquél en el que terminas de escribir la memoria. No sólo porque terminas un largo camino, sino también porque puedes tener unas palabras de agradecimiento para las personas que han estado contigo.

La primera de ellas es mi director de tesis, Rafael Corchuelo, ya que hubiera sido imposible terminar este trabajo sin su ayuda. Siempre ha confiado en mi trabajo y, desde que me dirigió el proyecto final de carrera, siempre ha tenido las palabras adecuadas en los momentos oportunos.

La ayuda de un grupo de investigación es fundamental para poder hacer realidad una tesis doctoral. En este sentido, The Distributed Group me ha dado el soporte y el ánimo necesario desde el comienzo de mi trabajo. En especial, Miguel Toro, por su buena predisposición desde el principio y por las revisiones realizadas. Sin su ayuda jamás hubiera empezado este trabajo. Además, las discusiones mantenidas con Joaquín Peña, David Ruiz y Antonio Ruiz, o los desayunos con José A. Pérez, Octavio Martín y David Benavides, han hecho que este trabajo sea mucho más llevadero.

Por último, pero no menos importante, le doy las gracias a mis padres y a mi mujer por haber aguantado mi mal humor, sobre todo los últimos meses de este trabajo en los que he estado al cien por cien dedicado a escribir esta memoria. Seguro que ellos tenían más ganas que yo mismo de que este momento llegara.

Sevilla
23 de Noviembre de 2004
J. L. A.

Resumen

You will have only an opportunity
to make a first impression.

Dicho popular

En los últimos años la web se ha consolidado como uno de los repositorios de información más importantes. Un gran reto para los agentes software ha sido tratar con esa cantidad, poco manejable de datos, para extraer información con significado. Este proceso es difícil por las siguientes razones: en primer lugar, la información en la web tiene como objeto su consumo por seres humanos y no contiene una descripción de su semántica, lo que ayudaría a los agentes entenderla; en segundo lugar, la web cambia continuamente, lo que tiene generalmente un impacto en la presentación de la información pero no en su semántica; por último, es un enorme repositorio con 7 500 Terabytes de información lista para ser consumida.

Los miembros de The Distributed Group han estado trabajando en sistemas distribuidos desde 1997. Concretamente, han trabajado en modelos de interacción multipartitos que proporcionan al programador de los mecanismos adecuados para describir interacciones complejas desde un punto de vista conceptual. Los resultados obtenidos se han materializado en publicaciones en revistas importantes y tesis doctorales. El trabajo de investigación en esta memoria abrió una nueva línea de investigación en el grupo. Su objetivo es facilitar el diseño e implementación de agentes software. Actualmente, esta línea de investigación se refuerza con la tesis de J. Peña, en la que se están desarrollando mecanismos para describir abstractamente las interacciones complejas en sociedades multi-agentes.

En esta memoria presentamos un nuevo marco de trabajo para la extracción de información con significado de la web sintáctica actual. Sus principales ventajas son: asocia semántica a la información extraída, mejorando la interoperabilidad del agente; trata los cambios en la web, potenciando la adaptabilidad; además, establece una separación de responsabilidades en la tarea de

extracción, automatizando el desarrollo de extractores de conocimiento distribuidos. Por último, el detallado estudio del trabajo relacionado demuestra que nuestra propuesta constituye una contribución original.

Parte I

Prólogo

Capítulo 1

Introducción

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction
of a new order of things.

Niccolo Machiavelli, 1469–1527
Italian dramatist, historian, and philosopher

En esta tesis doctoral se presenta un nuevo marco de trabajo de ayuda a los programadores en la construcción de agentes software capaces de entender el conocimiento que reside en la web actual. En este capítulo, primero introducimos algunos conceptos que definen el contexto de investigación en la Sección §1.1; después hacemos un resumen de la principales aportaciones en la Sección §1.2; por último, en la Sección §1.3, presentamos la estructura de esta memoria.

1.1. Contexto de investigación

En esta sección presentamos una visión general de los conceptos más importantes que usamos a lo largo de la memoria. Primero introduciremos el nuevo paradigma de los agentes software en la Sección §1.1.1; nuestra visión del concepto de ontología se presenta en la Sección §1.1.2; por último, la diferencia entre información y conocimiento se muestra en la Sección §1.1.3.

1.1.1. Agentes Software

Un agente software es una aplicación que exhibe las características descritas por Wooldridge y Jennings en [87], estas características son: autonomía, reactividad, proactividad y habilidad social. Autonomía significa que un agente funciona sin intervención directa de otros agentes o seres humanos y que tiene control sobre sus acciones y estado interno. Reactividad significa que un agente percibe su entorno y responde de manera oportuna a los cambios que ocurren en él. Proactividad significa que un agente no reacciona simplemente a los cambios en el entorno, sino que exhibe un comportamiento dirigido por objetivos, tomando la iniciativa cuando lo considera apropiado. Habilidad social significa que un agente interactúa con otros agentes (si es necesario) para completar sus tareas y ayudar a otros agentes a alcanzar sus objetivos. Muchos investigadores están de acuerdo con la visión de los agentes software como una caracterización y dependiendo de su área añaden nuevas características, por ejemplo: movilidad en sistemas distribuidos, o adaptabilidad en aprendizaje automático.

Los atributos autonomía, reactividad, proactividad y habilidad social no son booleanos. Es necesario pensar en ellos en términos de dimensión o grado de medida [60], de manera que un agente software es una aplicación que excede un predefinido umbral de dichos atributos. Por tanto, una aplicación para que sea considerada agente software no necesita exhibir un grado máximo de autonomía, reactividad, proactividad y habilidad social. Los atributos no son cuantificables y el ajuste del umbral es totalmente subjetivo. Así, en función del umbral fijado, un agente software puede ser desde un procedimiento sencillo con direcciones preceptivas, a aplicaciones de nueva generación que realmente exhiben capacidades de aprendizaje e inteligencia artificial.

Nwana presenta en [71] una clasificación del software existente basado en el paradigma de agentes. Define siete tipos diferentes: cooperativos, de interface, móviles, de información, reactivos, híbridos e inteligentes. También afirma que algunos de estos tipos se podrían ver como características en un

espacio multi-dimensional, permitiendo el desarrollo de sistemas heterogéneos de agentes. Para cada uno de estos tipos, Nwana presenta su motivación y principales características, así como algunos ejemplos de aplicación. En esta memoria, nos centramos en el estudio de los agentes de información.

Los agentes de información son agentes software que tienen acceso a múltiples, heterogéneas y geográficamente distribuidas fuentes de información, y manejan la información de interés a favor de sus usuarios u otros agentes. Entre las tareas que realizan se incluyen recuperar, extraer, analizar e integrar información. Estos agentes son frecuentemente usados para analizar las ofertas de competidores o para pronosticar actuaciones futuras en base a la información que reside en las páginas web [9, 29, 47], por ejemplo, BargainFinder compara los precios de CDS de distintas tiendas en Internet, Jaspe trabaja a favor de un usuario o comunidad de usuarios para almacenar, recuperar e informar a otros agentes de información de utilidad en la web. En esta memoria nos centramos en esta clase de agentes, es decir, agentes que necesitan la información que reside en la web para cumplir sus objetivos.

1.1.2. Ontologías

El término ontología ha sido usado desde hace mucho tiempo en filosofía, donde se refiere a una teoría filosófica sobre la naturaleza de la existencia. Este término fue redefinido más adelante en inteligencia artificial. Actualmente, existen muchas definiciones de ontología que ofrecen diferentes, pero complementarios, puntos de vistas de una misma idea "las ontologías son modelos del mundo". Guarino presentó en [35] un detallado estudio de las distintas definiciones del término ontología en el campo de la inteligencia artificial.

Actualmente, en inteligencia artificial, la definición de ontología más citada es de Gruber en 1993 [33, 34]:

An ontology is an explicit specification of a conceptualisation.

Esta definición fue refinada por Borst en 1997:

Ontologies are defined as a formal specification of a shared conceptualisation.

Studer y colegas en 1998 [84] analizan las definiciones anteriores y explican algunos conceptos que aparecen en ellas:

> *Conceptualization refers to an abstract model of phenomena in the world by having identified the relevant concepts of those phenomena. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine readable. Shared reflects that ontology should capture consensual knowledge accepted by the communities.*

Otra definición importante fue dada por Berners-Lee y colegas en el contexto de la web semántica [5]:

> *An ontology is a document or file that formally defines the relations among terms. A typical ontology has a taxonomy defining the classes and their relations and a set of inference rules powering reasoning functions.*

Dos ideas importantes se obtienen de esta definición: la primera es que, en la web semántica, una ontología es algo más que una mera taxonomía de conceptos; por último, que una ontología debería permitir el razonamiento en el dominio que describe.

Para los propósitos de esta tesis doctoral, una ontología es una descripción formal, explícita de conceptos y las relaciones entre ellos en un dominio del discurso, que nos permite especificar la semántica de la información que reside en la web. Una ontología se compone de los siguientes elementos: conceptos que representan entidades en el dominio, propiedades que describen relaciones entre conceptos o atributos de un concepto, restricciones en las propiedades y axiomas para modelar sentencias que son siempre verdades.

1.1.3. Información y conocimiento

Los conceptos información y conocimiento se han definido en varios contextos como: filosofía, tecnología de la información, ciencias sociales o economía, teniendo distintas connotaciones en cada uno de ellos. La definición que utilizamos en esta tesis doctoral se establece en el contexto de la web semántica, en este sentido, el modelo usado para representar la información o conocimiento es la web y los agentes software son los consumidores de dicha información o conocimiento.

Información son hechos, declaraciones acerca de un sujeto particular, sin una descripción explícita formal de su significado. La web es un modelo apropiado para la representación de información y HTML [75, 76] es el lenguaje usado para su publicación. Los documentos HTML son ficheros ASCII que

contienen etiquetas que se usan para especificar cómo los navegadores deben mostrar la información a seres humanos.

Conocimiento es información que un agente software puede entender. Para entender la información que reside en un sitio web, un agente software necesita una ontología que especifique el significado de la información que ofrece ese sitio y un conjunto de instancias de individuos que relacione dicha información con los conceptos especificados en la ontología (una ontología junto con un conjunto de instancias de individuos constituye una base de conocimiento). El término "entender" se refiere a la habilidad de razonar automáticamente a partir de ese conocimiento; existen distintos niveles de razonamiento, desde la clasificación de conceptos e individuos (taxonomía), hasta la adquisición de nuevo conocimiento a partir de los axiomas especificados en la ontología.

La Figura §1.1 muestra la diferencia entre información y conocimiento. Para ello ofrece dos vistas: la primera es información representada como un simple texto; esta representación no es apropiada para que los agentes software puedan automáticamente razonar, dado que no existe una especificación formal de la semántica de los conceptos que aparecen en el texto; para que un agente software pudiese razonar deberíamos embeber el conocimiento sobre el texto en la lógica funcional del agente. La otra vista es conocimiento y está compuesta por una ontología formulada en lógica de primer orden y un conjunto de instancias de individuos (que relaciona la información que nos ofrece la página web con la ontología); los agentes software pueden automáticamente razonar sobre esta base de conocimiento.

1.2. Resumen de aportaciones

Nuestra tesis doctoral se centra en dar soporte de ingeniería a los desarrolladores de agentes de información que necesitan el conocimiento que reside en la web para satisfacer sus objetivos. Hemos analizado las propuestas más importantes en extracción de información y extracción de conocimiento de la web y hemos concluido que no son apropiadas para el desarrollo, a un coste razonable, de agentes web que tienen la capacidad de "entender" la información que reside en la web, ésto es debido a tres problemas importantes: la web actual es sintáctica, lo que complica la comunicación e interoperabilidad entre agentes [5]; la estructura y el aspecto de las páginas web pueden cambiar inesperadamente, lo que puede invalidar los métodos automáticos de extracción [6, 27, 59, 83]; por último, la web es enorme y distribuida, lo que hace que soluciones manuales no sean apropiadas y justifica la necesidad de herramientas automáticas y distribuidas que permitan aprovechar todo el potencial que nos ofrece.

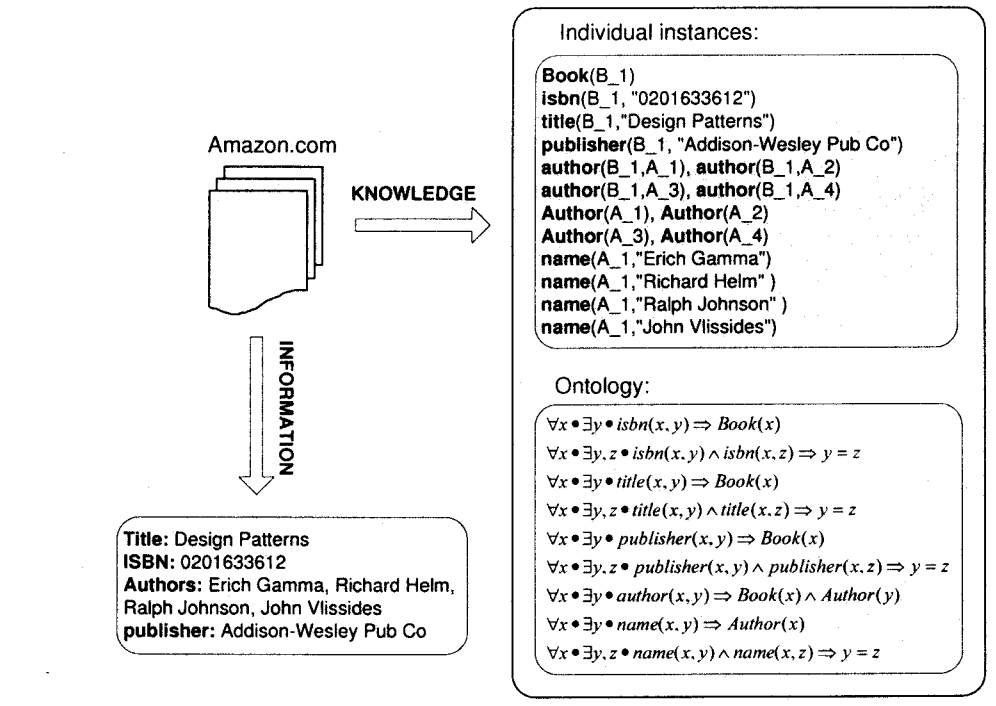


Figura 1.1: *Información vs. conocimiento.*

Nuestro principal resultado es un marco de trabajo que hemos llamado WebMeaning. Sus principales ventajas son: asocia semántica a la información extraída, mejorando la interoperabilidad del agente; trata los cambios en la web, potenciando la adaptabilidad; además, establece una separación de responsabilidades en la tarea de extracción, automatizando el desarrollo de extractores de conocimiento distribuidos. Por último, el detallado estudio del trabajo relacionado demuestra que nuestra propuesta constituye una contribución original.

1.3. Estructura

Esta memoria está organizada de la siguiente forma:

Parte I: Prólogo. Contiene sólo esta introducción.

Parte II: Estado del arte. Nuestro objetivo en esta parte es proporcionar al lector de un profundo entendimiento del contexto de investigación en el que nuestros resultados han sido desarrollados. En el Capítulo §2 presentamos la web actual, y a continuación la nueva web semántica; también se introducen los formalismos y lenguajes que posibilitan la transición. En el Capítulo §3 y §4 presentamos propuestas de dos dimensiones distintas, extracción de información y extracción de conocimiento.

Part III: Nuestra aportación. Es el núcleo de la tesis, y está organizada en cuatro capítulos. En el Capítulo §5 motivamos nuestra investigación y mostramos cómo las propuestas existentes en la actualidad no son lo suficientemente prácticas. En el Capítulo §6 presentamos nuestro marco de trabajo para la extracción de conocimiento de la web; WebMeaning divide el proceso de extracción de conocimiento en cuatro actividades: extracción de información, verificación de información, traducción de la información a conocimiento, y verificación semántica. En el Capítulo §7 definimos abstractamente nuestra propuesta para la traducción de la información a conocimiento. Por último, en el Capítulo §8 presentamos los tipos abstractos de datos y algoritmos necesarios para implementar el marco de trabajo; además demostramos que son correctos.

Part IV: Notas finales. Es un capítulo en el que resumimos las principales conclusiones y las líneas de investigación futuras.

Part V: Apéndices. La notación utilizada se encuentra resumida en el Apéndice §A. En el Apéndice §B probamos la equivalencia entre dos estructuras de datos de interés en esta memoria.

Parte II

Estado del arte

Capítulo 2

De información a conocimiento

Internet is so big,
so powerful and pointless
that for some people it is
a complete substitute for life.

George Carlin, 1937–
American comedian and actor

La web actual fue diseñada como una fuente de información para consumo humano. Un reto importante en la web del futuro es que las aplicaciones sean capaces de participar. Esto significa una transición de una web sintáctica a una web semántica, en la que meta-datos expresan el significado de la información que reside en la web. Este capítulo se organiza de la siguiente manera: en la Sección §2.2 introducimos algunos conceptos del campo de representación del conocimiento; en la Sección §2.3 discutimos las características de la web actual; a continuación, la web semánica es presentada en la Sección §2.4; finalmente, la Sección §2.5 presenta brevemente las estrategias para razonar en la web semántica.

2.1. Introducción

La web actual es un repositorio de información. Los consumidores de esta información son los humanos, que son conscientes de su valor. Según el Centro de UCLA para la Política de Comunicación, la web está considerada como una valiosa fuente de información por la inmensa mayoría de personas conectadas a Internet; en el 2002, el 60,5 por ciento de todos usuarios consideraron que Internet era una importante fuente de información [26]. La idea de tener un repositorio completamente accesible provocó que investigadores comenzasen a trabajar en propuestas cuyo objetivo era gestionar la información en la web [2, 25]. Actualmente, gestionar esta información es un gran desafío debido a que la web es un repositorio sintáctico, dinámico, masivo y distribuido de documentos estructurados, semi-estructurados y sin estructura.

La web actual está cambiando a un repositorio de información con significado que los agentes software son capaces de entender, lo que posibilita que realicen acciones complejas para sus usuarios con menos intervención humana [5]. Este cambio requiere mucho trabajo de investigación y la integración de tecnologías de diferentes campos como ingeniería del software o inteligencia artificial. Las Ontologías [14] juegan un papel importante en la visión de una web semántica consumida por agentes software, proporcionándonos la representación de modelos conceptuales de un dominio particular que se puede compartir y comunicar entre humanos y agentes. Es decir, las ontologías nos permiten especificar información semántica acerca de la información en la web.

2.2. Representación del conocimiento

En esta sección preliminar, daremos una breve descripción de los formalismos y lenguajes de uno de los campos centrales de inteligencia artificial llamado representación del conocimiento. La investigación en representación del conocimiento se centra en el desarrollo de lenguajes y formalismos altamente expresivos para representar bases de conocimiento y en la forma de razonar con ellos. La diferencia entre formalismo y lenguaje se encuentra en el nivel de abstracción que usan para representar el conocimiento. Los formalismos, también conocidos como paradigmas, se encuentran a un nivel bajo de abstracción, siendo la base de los lenguajes para representar conocimiento; Por ejemplo, los marcos y la lógica de primer orden son dos formalismos que se combinan en el lenguaje Ontolingua [21] con objeto de proveer de un entorno de trabajo distribuido y colaborativo para la gestión de ontologías.

Es necesario indicar que existe una importante diferencia entre la noción de ontología y el formalismo o lenguaje que la expresa. Diferentes formalismos o lenguajes de diferentes comunidades pueden ser usados para expresar un mismo modelo conceptual. Por ejemplo, en el área de base de datos, se utilizan modelos para representar la estructura lógica de la información, con el objetivo de almacenarla y recuperarla de forma eficiente; los ingenieros del software también se ayudan de modelos para representar el dominio del problema, con el objetivo de construir aplicaciones. Sin embargo, en ninguno de los campos anteriores, el razonamiento sobre los modelos juega un papel secundario.

2.2.1. Formalismos

Los formalismos de representación de conocimiento se clasifican en no-lógicos y lógicos. Los formalismos no lógicos fueron desarrollados motivados por la idea de que la lógica no es apropiada para representar conocimiento; están basados en experimentos cognitivos acerca de cómo se almacena el conocimiento en el cerebro humano. Por otra parte, los formalismos lógicos se basan en la lógica para representar conocimiento.

Las redes semánticas, los sistemas de marcos y los grafos conceptuales son los formalismos no-lógicos más importantes en inteligencia artificial:

Redes semánticas. Las redes semánticas fueron introducidas en 1967 por Quillian [78]. Es una notación para representar conocimiento en patrones de vértices interconectados por ejes etiquetados (gráfo dirigido). Los vértices representan conceptos o individuos y los ejes son relaciones entre conceptos o propiedades asociadas a los conceptos. Este formalismo fue criticado por no disponer de una semántica bien definida.

Sistemas de marcos. Minsky introdujo los sistemas de marcos en 1975 [67]. Un marco es una estructura de datos para representar un concepto o una situación de manera orientada a objetos; un marco está compuesto de slots (o atributos), donde cada slot puede tener asociado descripciones o procedimientos; un conjunto de marcos interconectados recibe el nombre de sistema de marcos. La semántica de la parte monotónica o declarativa de éste formalismo ha sido descrita en lógica de primer orden, sin embargo, la parte no declarativa o no monotónica no tiene una semántica bien definida.

Grafos conceptuales. los grafos conceptuales fueros introducidos por Sowa en 1984 [81]. Es el formalismo más popular para representar gráfica-

mente conocimiento y pueden ser vistos como descendientes de los sistemas de marcos y de las redes semánticas. Los grafos conceptuales son grafos etiquetados donde existen nodos de tipo conceptos conectados por otros nodos de tipo relación. La semántica formal de éste formalismo está basada en lógica de primer orden.

La lógica de primer orden, la descriptiva y la no monotónica son los formalismos lógicos más populares en inteligencia artificial:

Lógica de primer orden. Es el formalismo más importante y expresivo para representar conocimiento. Permite representar hechos sobre un dominio de discurso e inferir conclusiones que garantizan que, si los hechos iniciales eran verdaderos entonces las conclusiones son también verdaderas. Sus características principales son que es un lenguaje formal bien conocido, con una sintaxis y semántica bien definidas. Sin embargo, la lógica de primer orden es semi-decidible, lo que hace que el problema de inferencia sea computacionalmente no tratable.

Lógicas descriptivas. Son una familia de lenguajes para la representación de conocimiento con una semántica formal bien formulada y basada en lógica de primer orden [3]. Éste formalismo está considerado como un formalismo unificador para la representación estructurada de conocimiento, por ejemplo, los marcos, las representaciones orientadas a objetos o los diagramas entidad-relación pueden ser formulados con lógicas descriptivas. Una base de conocimiento representada en lógica descriptiva se divide en dos partes: la Tbox que define la estructura del dominio y la Abox que describe un ejemplo concreto del dominio. La semántica formal de los lenguajes descriptivos es especificada usando lógica de primer orden, por tanto, las lógicas descriptivas pueden ser vistas como fragmentos de lógica de primer orden.

Lógicas no monotónicas. el término "no monotónico" cubre una familia de formalismos que tienen como objetivo capturar y representar inferencia no factible [56]. Éste tipo de inferencia permite a los razonadores llegar a conclusiones de forma tentativa, reservándose el derecho de retractase de ellas en base a nueva información. Existen tres aproximaciones importantes para formalizar el razonamiento no monotónico: circunscripción, lógica por defecto y lógica modal.

2.2.2. Lenguajes tradicionales

Los lenguajes para representar conocimiento se clasifican en lenguajes tradicionales [13] y lenguajes ontológicos para la web. Ésta clasificación viene mo-

tivada por la web semántica, que justifica el desarrollo de nuevos lenguajes que tienen como objetivo especificar la semántica de la información que contienen las páginas webs y en el intercambio de ontologías en el entorno distribuido que ofrece la web. Aquí presentamos un resumen de los lenguajes tradicionales más importantes, los lenguajes ontológicos para la web serán introducidos más tarde, cuando se presente la web semántica.

CARIN. Es una familia de lenguajes, cada uno de ellos combina una lógica descriptiva $\mathcal{L}$ con reglas de Horn [58]. Un lenguaje concreto en CARIN viene representado por CARIN-$\mathcal{L}$. Una base de conocimiento en CARIN-$\mathcal{L}$ está formada por tres componentes: una terminología en lógica descriptiva, un conjunto de reglas de Horn y un conjunto de instancias usando el vocabulario definido en la terminología. Por último, existen algoritmos para razonar sobre bases de conocimiento definidas en CARIN.

Frame logic. Integra los sistemas de marcos con la lógica de predicados para especificar bases de datos orientadas a objetos, sistemas de marcos y programas lógicos [53]. Su principal objetivo es integrar distintos constructores para el modelado conceptual como clases, atributos, herencia o axiomas. Frame logic tiene una semántica bien definida y existen algoritmos basados en pruebas lógicas para razonar sobre las bases de conocimiento.

LOOM. Es un lenguaje y entorno para la construcción de aplicaciones inteligentes [62]. El núcleo de LOOM es un sistema para la representación de conocimiento basado en lógica descriptiva y reglas de producción. El conocimiento en LOOM viene representado por definiciones, reglas, hechos y reglas por defecto. LOOM usa un motor deductivo con encadenamiento hacia delante, unificación semántica y orientación a objetos para dar soporte al procesamiento de preguntas.

OCML. Soporta la construcción de bases de conocimiento por medio de varios tipos de constructores [68]. Permite la especificación de funciones, relaciones, clases, instancias y reglas. Con objeto de hacer mas eficiente el razonamiento con las bases de conocimiento añade algunos mecanismos lógicos extras. También ofrece los mecanismos para lanzar consultas sobre el conocimiento expresado en OCML.

Ontolingua. Proporciona un entorno colaborativo y distribuido para navegar, crear, editar, editar, modificar y usar ontologías [21]. El lenguaje utilizado para representar conocimiento está basado en el sistema de marcos y en lógica de primer orden formulada en KIF [31]. Por último, existe un motor de inferencia para razonar sobre las bases de conocimiento especificadas en Ontolingua.

2.3. La web actual

La web actual fue definida por la W3C como[4]:

It is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

La web ha superado todas las expectativas iniciales y no solamente es el mayor repositorio de información accesible universalmente, sino que también ha tenido importantes repercusiones en la sociedad. Un buen ejemplo de ésto es el portal temático de la Europe's Information Society* que define un conjunto de actividades que tienen como objetivo ofrecer servicios gubernamentales, de sanidad, aprendizaje a los ciudadanos europeos.

Tener un repositorio accesible de información propició que investigadores de base de datos [25] e inteligencia artificial [2] comenzasen a trabajar en propuestas para gestionar la información residente en la web. Sin embargo, algunas de las características de la web hacen difícil esta tarea [39]:

La web es sintáctica. Los documentos HTML son archivos ASCII con etiquetas que precisan la forma en la que los navegadores deben presentar la información. No existe, por tanto, referencia alguna a la información semántica que describe el significado de la información que reside en esos archivos.

La web es dinámica. Está continuamente cambiando [6, 59]. Francisco-Revilla et al. [27] clasificaron los posibles cambios que se dan en la web en: de contenido o semánticos, de presentación, estructurales y de comportamiento. El primero de ellos se refiere a modificaciones del contenido de la página desde punto de vista del usuario; el segundo son los cambios relacionados con la representación del documento que no afecta al contenido; el tercero se refiere a los enlaces del documento con otros documentos; el último se refiere a modificaciones en los componentes activos del documento (scripts, plugings o applets).

La web es enorme y distribuida. Es imposible verificar el número exacto de páginas web en la WWW, no obstante, existen algunas estimaciones de consultoras. Estas estimaciones diferencian dos clases de páginas web. Las primeras se encuentran en la superficie de la web y son todas aquellas páginas estáticas y accesibles públicamente. Las otras se encuentran

*http://europa.eu.int/information_society/eeurope/2005/index_en.htm

constituyen la web profunda y son todas aquellas que se construyen dinámicamente a partir de bases de datos. Según estimaciones de Cyveillance **, la cantidad de información en la superficie de la web es de 25 a 50 Terabytes y 7.500 Terabytes para la web profunda.

Toda esa información no está centralizada. Desde un punto de vista abstracto, la web es un grafo dirigido cuyos nodos son páginas web y cuyos ejes son conexiones. Cada página web puede residir en una máquina remota, lo que hace de la web un sistema distribuido.

2.4. La web semántica

La web semántica fue definida en Berners-Lee et al. [5] como:

The semantic web is an extension to the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

La web semántica es una extensión de la actual. Esta extensión supone anotar las páginas web con meta-datos que describen la semántica de la información que reside en ellas. La web semántica simplificará y mejorará las técnicas de extracción de información tremendamente. No obstante, anotar todas las páginas web existentes requiere de mucho esfuerzo.

Los lenguajes ontológicos para la web posibilitan la especificación de ontologías y la anotación de las páginas web con semántica. Una página web semántica nos ofrece dos vistas diferentes: la primera es para consumo humano, un documento en formato HTML que los navegadores convierten a un documento multimedia (la web actual); la segunda es para consumo de los agentes software, un documento estructurado con anotaciones semánticas (referencias a conceptos y propiedades definidas en una ontología externa) de la información de interés.

Los lenguajes ontológicos para la web están influenciados por el campo de la ingeniería del conocimiento. No obstante, difieren de los formalismos tradicionales en que están centrados en la web. Las principales características de los lenguajes ontológicos para la web son las siguientes [40]:

1. Extienden los estándares de la web existentes, como HTML, XML, RDF-S, permitiendo su integración con otras tecnologías de la web.

**Cyveillance.com

Language	Score	Normalised score	Underlying formalism
XOL	2.5	33.33	Frame systems
SHOE	-1.5	3.70	Frames systems
OML	10.5	92.59	Conceptual graphs
RDF-S	-2	0	Semantic networks
OIL	11.5	100	Description logics
DAML+OIL	11.5	100	Description logics
OWL	11.5	100	Description logics

Cuadro 2.1: *Expresividad de los lenguajes ontológicos para la web.*

2. Posibilitan la definición de diversas ontologías, potencialmente conflictivas, con objeto de trabajar con una web descentralizada.

3. Posibilitan la evolución de los vocabularios a medida que el entendimiento humano mejora, con objeto de contemplar la rápida evolución de la web.

4. Son escalables, con objeto de tratar el gran tamaño de la web.

5. Están formalmente especificados y proveen de soporte de razonamiento automático, con objeto de ser de utilidad para los agentes software.

6. Tienen un nivel de expresividad "adecuado" para representar el conocimiento residente en la web.

Existen muchos lenguajes ontológicos para la web, como: SHOE [61], RDF-S/RDF [7], XOL [50], OML [52], DAML-ONT [65], OIL [23, 24], DAML+OIL [42, 64] o OWL [17]. La Tabla §2.1 resume los resultados del análisis presentado en Ref. [32] que fue llevado a cabo para comparar sus principales características. Un valor de 1 se le ha asignado a cada característica soportada por el lenguaje, 0,5 si la soporta parcialmente y -1 si no la soporta. El resultado de esta conversión numérica se muestra en columna "score" y su normalización en la columna "normalised score". Fácilmente se observa que los lenguajes con mejor puntuación son: OIL, DAML+OIL y OWL.

A continuación, presentamos una breve descripción de los lenguajes más importantes en la evolución de los lenguajes ontológicos para la web (Figura §2.1). Éstos son: RDF-S/RDF, SHOE, DAML+OIL y OWL.

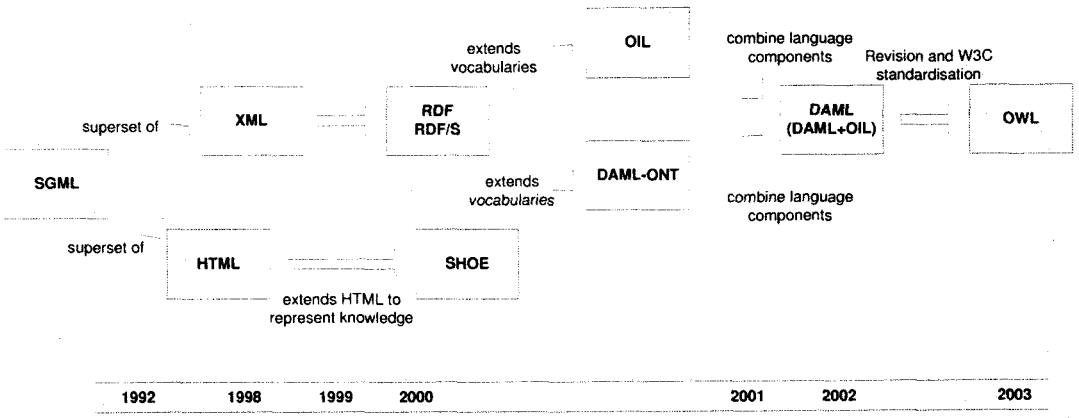


Figura 2.1: *Evolución de los lenguajes ontológicos para la web.*

RDF and RDF-S. Fueron desarrollados por la W3C con el propósito de dar una forma estándar de especificar datos acerca de algo (RDF) y su interpretación (RDF-S) [7]. La principal característica de RDF es que es usado como un marco de trabajo general para la integración e intercambio de conocimiento descrito en lenguajes ontológicos más expresivos, por ejemplo, OIL, DAML+OIL y OWL extienden RDF-S añadiendo primitivas de modelado mientras mantienen el máximo número de constructores RDF-S con objeto de mantener compatibilidad entre ellos.

El modelo de datos RDF [77] está basado en el formalismo de redes semánticas. La semántica del modelo de datos está definida axiomáticamente, traduciendo el modelo de datos RDF a sentencias en lógica de primer orden y dando los axiomas que restringen su posible interpretación.

SHOE. Fue uno de los primeros intentos para definir un lenguaje ontológico para la web [61]. SHOE trata los problemas de un entorno dinámico y distribuido donde las ontologías están interrelacionadas y sujetas a cambios. Es por eso que define un conjunto de directivas (etiquetas HTML) que permiten especificar la versión, compatibilidad, información acerca de ontologías importadas y el renombrado local de constantes importadas. Una ontología en SHOE es un documento HTML compuesto de: las directivas anteriores, una jerarquía para clasificar las instancias, propiedades que describen relaciones entre conceptos o atributos de conceptos y reglas de inferencia especificadas como reglas Horn.

DAML+OIL. Es el esfuerzo de una unión entre los lenguajes DAML-ONT desarrollado por DARPA y OIL desarrollado en el contexto del proyec-

to europeo IST OnToKnowledge [42, 64]. DAML+OIL sigue una aproximación de marcos para la descripción del dominio, es decir, es posible usar clases y propiedades. Una ontología consiste en un conjunto de axiomas que establecen relaciones entre clases y propiedades. Formalmente, DAML+OIL es la serialización sintáctica de una lógica descriptiva [43], donde una ontología se corresponde a la descripción de una terminología (Tbox).

OWL. Es el lenguaje ontológico para la web recomendado por la W3C [17]. Ha sido diseñado como una revisión de DAML+OIL y como tal combina las primitivas de modelado de los formalismos basados en marcos, con la semántica formal y la posibilidad de razonamiento de la lógica descriptiva. De acuerdo con la recomendación de la W3C, OWL provee de tres sub-lenguajes diseñados para ser usados por comunidades específicas de desarrolladores y usuarios: OWL Lite para dar soporte a aquellos usuarios que requieren poca expresividad (clasificación jerárquica de los conceptos y simple restricciones); OWL DL para aquellos usuarios interesados en una máxima expresividad mientras existen garantías computacionales de completitud y decidibilidad; y OWL Full para aquellos usuarios que quieren la máxima expresividad si garantías computacionales.

2.5. Rasonamiento en la web semántica

Los lenguajes ontológicos para la web permiten representar el conocimiento que reside en la web independientemente de la aplicación que lo consume, pero dependiente del dominio. Los razonadores realizan complejas tareas con ese conocimiento, por ejemplo, chequear la consistencia de la ontología, verificando la sintaxis y el uso de la ontología para garantizar que los individuos cumplen todas las restricciones impuestas en la ontología; comprobar la satisfabilidad; o procesar preguntas formuladas por usuarios humanos o agentes software. Estas tareas son llevadas a cabo utilizando distintas estrategias. Los razonadores basados en lógica descriptiva se enfocan hacia la computabilidad, lo que permite la definición de decidibles y eficientes algoritmos mientras se mantiene un alto grado de expresividad. Los razonadores basados en lógica de primer orden tratan con lenguajes más expresivos pero no dan garantías de computabilidad.

La Tabla §2.2 resume las ideas anteriores analizando los razonadores más usados: cwm, Fact, F-OWL, Pellet, RACER, Surnia, TRIPLE, Hoolet y XSB. Debido a la similaritud entre DAML+OIL y OWL, usamo OWL para referirnos

a ambos lenguajes. Por último, indicar que se deduce de esta tabla que éste campo necesita todavía de mucho trabajo de investigación.

	cwm	Fact	F-OWL	Pellet	RACER	Surnia	TRIPLE	Hoolet	XSB
Based on	Horn	DL	Horn, Frame, Higher order	DL	DL	FOL	Horn, DL	FOL	Horn
Support	RDF	OWL-DL	OWL-Full	OWL-DL	OWL-DL	OWL-Full	RDF	OWL-DL	SHOE
Complete consistency checker	No	No	No	OWL-Lite	OWL-Lite	No	No	No	No
Decidable	No	Yes	No	Yes	Yes	No	Yes	No	No
Query			RDQL	RDQL	Racer query language		Horn logic style		Horn logic style
Known limitation	RDF	No Abox support	Poor scaling			Poor scaling		Poor scaling	SHOE

Cuadro 2.2: *Características de los razonadores para la web semántica.*

Capítulo 3

Extracción de información de la web

The beginning of knowledge
is the discovery of something
we do not understand.

Frank Herbert, 1920–1986
American science fiction novelist

La web actual es una enorme y valiosa fuente de información. En este Capítulo se presentan las propuestas que tienen como objetivo la extracción de la información que reside en la web. Está organizado como sigue: en la Sección §3.1 introducimos el capítulo; en la Sección §3.2 presentamos algunos atributos que nos permiten caracterizar las propuestas en extracción de información; a continuación, las propuestas más importantes son presentadas en la Sección §3.3; por último, en la Sección §3.4 presentamos la necesidad del mantenimiento de los sistemas extractores de información para hacer frente a los cambios en la web.

3.1. Introducción

Los algoritmos utilizados para la extracción de información de la web se llaman wrappers [19]. Los wrappers pueden ser codificados manualmente, usando propiedades de la página web, buscando normalmente cadenas que delimitan los datos que se quieren extraer. La generación manual de wrappers es el enfoque escogido por propuestas en el área de las bases de datos* como TSIMMIS [10], ARANEUS [66] o JEDI [46]. El objetivo que persiguen estas propuestas es el de integrar fuentes heterogéneas de información como las bases de datos tradicionales y las páginas web para que el usuario pueda trabajar en ellos como si fueran una fuente homogénea de información. Una contribución importante en el campo de la extracción de información la proporcionó Kushmerick. El introdujo mecanismos inductivos y definió una nueva clase de wrappers llamados wrappers inductivos. Estos algoritmos son componentes que utilizan reglas de extracción generadas automáticamente, para ello hacen uso de técnicas como: programación lógica inductiva, métodos estadísticos o gramáticas inductivas. Estas reglas configuran un algoritmo genérico para la extracción de información de un sitio web. La figura §3.1 muestra el ciclo de vida de un wrapper inductivo.

Los wrappers tienen un determinado período de vida necesitando de mantenimiento. El mantenimiento de wrappers consiste en dos tareas: verificación y reconstrucción. Un sistema de verificación de wrappers comprueba si el wrapper está funcionando correctamente; un wrapper funciona incorrectamente si la información extraída no es la esperada y esta situación es causada por cambios en la web. La reconstrucción de wrappers consiste en la construcción de un nuevo wrapper cuando dejan de funcionar correctamente. Por último destacar que existe pocas propuestas en la literatura para resolver el problema del mantenimiento de los wrappers.

3.2. Caracterizando los wrappers

En la literatura de extracción de información, los términos: record, frame, y template se refieren a una misma idea: una estructura de datos para representar la información extraída, que ésta compuesta de slots o atributos; además, los términos slot y atributo se utilizan indistintamente para referirse a una característica extraída. En esta memoria, usamos los términos de marco y atributo. A continuación presentaremos algunas características que nos permitirá

*En el área de las Bases de Datos, los algoritmos para la extracción de información se llaman mediadores de información.

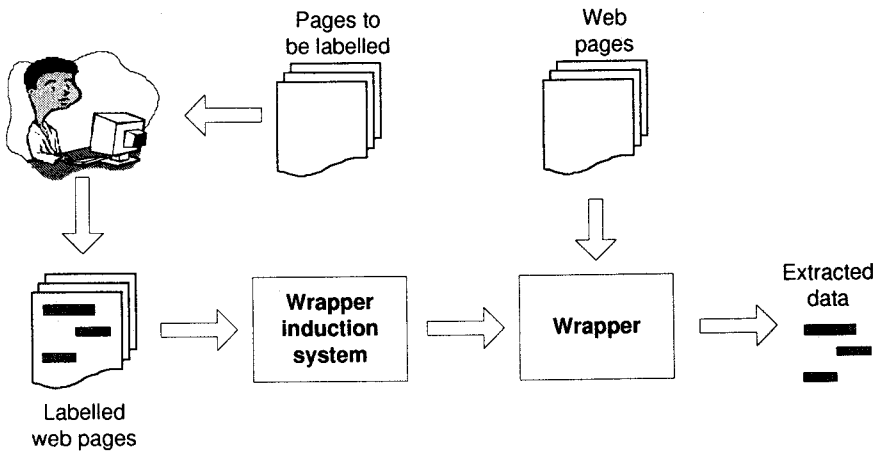


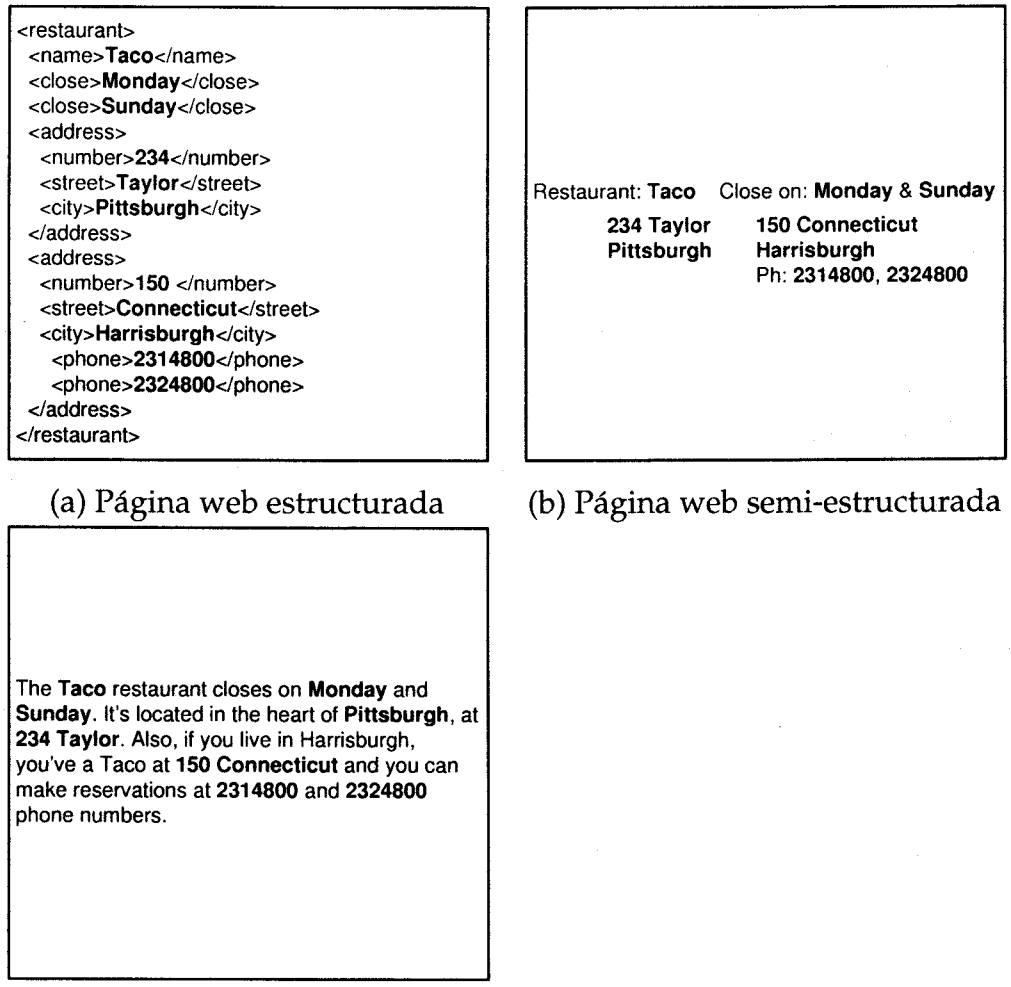
Figura 3.1: *Ciclo de vida de un wrapper inductivo.*

definir el alcance de un wrapper (el número de páginas webs que pueden ser manipuladas).

Extracción single-slot vs. multi-slot. Una wrapper realiza extracción single-slot cuando sólo puede extraer atributos aislados de una página web. Un wrapper realiza extracción multi-slot si puede asociar los atributos extraídos en marcos.

Páginas web estructuradas, semi-estructuradas y sin estructura. Hsu y Dung [45] presentaron una clasificación de las páginas web basada en las características estructurales de los atributos a extraer [45]. La clasificación distingue tres tipos de páginas webs: una página web es estructurada si proporciona información con un formato predefinido y estricto, de manera que cada atributo en una tupla se puede extraer correctamente en base a propiedades sintácticas; una página web es semi-estructurada si contiene atributos opcionales, multivaluados o permite permutaciones en el orden en el que aparecen los atributos; una página web es sin estructura si se requiere conocimiento lingüístico para extraer los atributos. La figura §3.2 muestra un ejemplo de página web estructurada, semi-estructurada y sin estructura.

Extracción de información tabular vs. jerárquica. Los wrappers también se pueden clasificar en cómo la información se estructura en las páginas web. La mayoría de los wrappers extraen información estructurada de manera relacional o tabular. Hay algunos que son también capaces de extraer información estructurada jerárquicamente, es decir, los valores de los atributos se presentan en forma de árbol en lugar de tabularmente.



(a) Página web estructurada

(b) Página web semi-estructurada

(c) Página web sin estructura

Figura 3.2: *Página web estructurada, semi-estructurada, y sin estructura.*

Name	Struc.	Semi-struc.	Unstruc.	S-slot	M-slot	Hi.
Rapier	✓	✓		✓		
SRV	✓	✓		✓		
WHISK	✓	✓	✓	✓	✓	
WIEN	✓			✓	✓	
SoftMealy	✓	✓		✓		
STALKER	✓	✓		✓		✓

Cuadro 3.1: *Resumen de las características de los wrappers inductivos.*

3.3. Wrappers inductivos

A continuación presentamos los wrappers inductivos más importantes: RAPIER, SRV, WHISK, WIEN, SoftMealy y STALKER. La tabla §3.1 resume las características de dichos wrappers. Las primeras tres columnas indican el tipo de páginas web que soportan; las siguientes dos columnas indican si hacen una extracción single-slot o multi-slot; la última columna indica si el wrapper es capaz de extraer información jerárquica.

RAPIER. Los algoritmos inductivos usados en RAPIER están basados en varios sistemas de programación lógica inductiva [8]. Las reglas de extracción generadas están basadas en delimitadores sintácticos de la información a extraer y en información semántica que describe el contenido. Están indexadas por un nombre de marco y un nombre de atributo, y tiene tres partes: un patrón denominado *pre-filler* que indica que debe de cumplir el texto que precede al atributo a extraer, un patrón *filler* que describe la estructura del atributo, y un patrón *post-filler* que indica que debe de cumplir el texto que sigue al atributo a extraer.

SRV. Genera reglas de extracción en lógica de primer orden, basadas en características que son simples (longitud, tipo de caracter, ...) o relacionales (*next – token*, *prev – token*: siguiente y anterior token al atributo a extraer, ...) [28]. Al usar lógica de primer orden sus reglas son muy expresivas. El aprendizaje consiste en identificar y generalizar las características encontradas en los ejemplos de prueba.

WHISK. Genera reglas basadas en expresiones regulares que tienen dos componentes: una que describe el contexto que hace que un atributo sea relevante, y otra que especifica los delimitadores exactos del atributo [80]. La

generación de los componentes anteriores depende del tipo de páginas web: para páginas sin estructura, sólo se usan patrones que describen el contexto; para páginas semi-estructuradas se usan patrones delimitadores; y finalmente, para páginas estructuradas se usan ambos.

WIEN. Introduce varios tipos de wrappers que asumen que los atributos a extraer siguen siempre un orden fijo y conocido, y que las páginas web tienen una organización HLRT [55]. Esto significa que es posible encontrar un delimitador de inicio de la información (Head), un conjunto de delimitadores por la izquierda y derecha para cada atributo (Left y Right), y un delimitador para indicar el final de la información. Las reglas de extracción generadas por WIEN son muy parecidas a las generadas por WHISK, con la salvedad de que WIEN sólo usa delimitadores que preceden y siguen a los atributos.

SoftMealy. Genera wrappers que son autómatas finitos no deterministas [44]. Los estados del autómata representan los atributos a extraer y las transiciones entre estados representan reglas contextuales que definen los separadores de los atributos. SoftMealy permite la extracción de páginas web en las que existen atributos opcionales o permutaciones.

STALKER. Extrae información jerarquizada de páginas web estructuradas y semi-estructuradas[69]. Para la extracción de información jerarquizada usa el formalismo Embedded Catalog Tree (ECT), que está basado en árboles. ECT permite especificar un esquema de salida que guía la tarea de extracción. Por cada vértice en el ECT, STALKER genera reglas de extracción, más una regla de iteración si el atributo que representa puede aparecer repetido en la página web.

3.4. Mantenimiento de los wrappers

Los wrappers se estropean debido a cambios en las páginas web. Francisco-Revilla et al. [27] presentaron una clasificación de los posibles cambios que se dan en la web. Usaremos esta clasificación para mostrar cuales de ellos pueden causar que un wrapper deje de funcionar correctamente:

De contenido o semánticos. Son cambios que sufre el contenido de la página web desde el punto de vista del usuario. Estos cambios pueden estropear a los wrappers, por ejemplo, `Amazon.com` cambió la forma de presentar los autores de los libros añadiendo la preposición "by" antes del nombre del primer autor; un wrapper afectado por este cambio, podría extraer "by Erich Gamma" en lugar de "Erich Gamma".

De presentación. Son cambios relacionados con la presentación de la información que no afectan al contenido. Estos cambios pueden estropear un wrapper que use reglas basadas en la estructura de la página web.

Estructurales. Son cambios en las conexiones entre páginas web (cambios en los enlaces). Estos cambios no suelen afectar a los wrappers ya que estos no tiene en cuenta los enlaces de las páginas web para generar las reglas de extracción.

De comportamiento. Son cambios en los componentes activos de las páginas web (scripts, plug-ins y applets). Estos cambios no afectan al funcionamiento de los wrappers ya que estos ignoran el contenido activo de las páginas web.

El mantenimiento de wrappers consiste en dos tareas: verificación y reconstrucción. La verificación la llevan a cabo los verificadores de wrappers, para ello toman la información extraída por el wrapper y deciden si está funcionando correctamente. La reconstrucción la realizan los generadores de wrappers, construyendo nuevos wrappers cuando se estropean. El wrapper nuevo puede ser reconstruido manualmente, o semi-automáticamente aplicando técnicas inductivas con nuevos ejemplos, o automáticamente construyendo los ejemplos y aplicando técnicas de re-inducción. La Figura §3.3 muestra un sistema de wrappers que usa un verificador y que automáticamente genera el wrapper cuando este se estropea.

Existen tan solo dos propuestas, que conozcamos, que tiene como objetivo solucionar el problema del mantenimiento de wrappers: RAPTURE y DataProG. La primera trata la tarea de la verificación de wrappers. La segunda trata tanto la verificación como la reconstrucción.

RAPTURE. Describe cada atributo de la información a extraer por un conjunto de valores estadísticos (longitud media de la palabra, número de palabras, número de caracteres especiales, número de dígitos ..) para caracterizar la información extraída [54]. Aprende los parámetros de las distribuciones normales que describen esos valores a partir de ejemplos de información extraída correctamente. Las probabilidades individuales de cada atributo se combinan para producir una probabilidad general que indique si el wrapper extrae la información correctamente.

DataProG. Lleva a cabo la verificación de wrappers aplicando técnicas de aprendizaje automático, aprendiendo las características de los atributos extraídos a partir de datos extraídos correctamente por el wrapper [57].

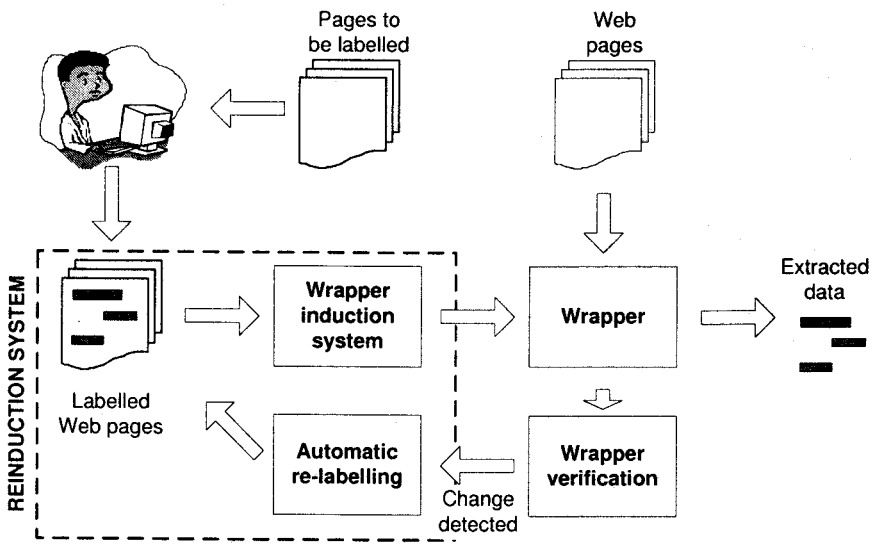


Figura 3.3: *Ciclo de vida del mantenimiento automático de wrappers inductivos.*

Entre las características se incluyen patrones que describen el comienzo y final de los atributos. Durante la fase de verificación, usa el wrapper para generar un nuevo conjunto de ejemplos de las mismas páginas webs que las usadas en la fase de entrenamiento; a continuación, calcula las características asociadas con los atributos de los ejemplos, si las distribuciones de ambos conjuntos de características son estadísticamente parecidas, el wrapper estará extrayendo la información correctamente; de lo contrario estará invalidado. DataProG realiza la re-inducción del wrapper a partir de un algoritmo automático de etiquetado, que usa un conjunto de patrones para identificar ejemplos de los datos a extraer en páginas web.

Capítulo 4

Extracción de conocimiento de la web

The beginning of knowledge
is the discovery of something
we do not understand.

Frank Herbert, 1920–1986
American science fiction novelist

También, la web actual es una enorme y valiosa fuente de conocimiento. En este capítulo clasificamos y presentamos propuestas que tienen como objetivo la extracción del conocimiento que reside en la web. El capítulo está organizado como sigue: la Sección §4.1 introduce el capítulo y clasifica las propuestas; los capítulos §4.2, §4.3 y §4.4 presentan las propuestas.

4.1. Introducción

Los sistemas de extracción de conocimiento tratan con los aspectos semánticos de la información que reside en la web. El trabajo de investigación de esta área se puede clasificar en tres categorías: extracción de ontologías, extracción de instancias de individuos y extración de bases de conocimiento. Los primeros extraen los modelos (ontologías) que dan semántica a la información en la web. Los segundos extraen un conjunto de instancias de individuos que relacionan la información en la web con los conceptos especificados en una ontología. Los últimos extraen la ontología e instancias de individuos.

La tabla §4.1 resume las características de las propuestas estudiadas. Las primeras dos columnas indican la clase de sistema y su principal objetivo; la tercera columna es el nombre de la propuesta; la cuarta identifica los tipos de páginas web soportados; la última columna indica los métodos usados. Por último, indicar que los sistemas para la extracción de ontologías no están directamente relacionados con nuestro trabajo, no obstante, para tener una visión global de los sistemas extractores de conocimiento, serán presentados sin entrar en detalles.

4.2. Sistemas de extracción de ontologías

Los sistemas de extracción de ontologías están enfocados a descubrir la información semántica en las páginas web. Según el objetivo que siguen, estos sistemas se clasifican en: aprendizaje de ontologías y aprendizaje de esquemas.

El aprendizaje de ontologías trata con la extracción del conocimiento ontológico de fuentes de información en las que este conocimiento es implícito. Entre las propuestas más importantes se encuentran WebOntEx, Text-To-Onto, ASIUM y, INTHELEX.

WebOntEx. Extrae ontologías de páginas web semi-automáticamente analizando páginas web sin estructura y semi-estructuradas en un mismo dominio de aplicación [36]. El módulo más importante de WebOntEx es un analizador heurístico que hace uso de programación lógica inductiva para clasificar los conceptos en tipos de entidad, obtener relaciones entre conceptos, atributos y taxonomías.

Text-To-Onto. Es un marco de trabajo para la extracción y mantenimiento de ontologías residentes en páginas web sin estructura [63]. Usa técnicas

Purpose	Objective	Proposal	Domain	Methods
Ontology extraction	Ontology learning	WebOntEx	Unstruc., semi-struc.	Inductive logic programming
		Text-to-Onto	Unstruc.	NLP and Data mining
		ASIUM	Unstruc.	Conceptual and hierarchical clustering
		INTHELEX	Unstruc.	Inductive logic programming
	Schema learning	Xtract	Struc.	Decomposition and factorisation of expressions
		Treeminer	Semi-struc.	Search on trees
		G. Cong et al.	Semi-struc.	Search on trees
Instances extraction	Question answering	Start/Omnibase	Unstruc.	NLP
		Squeal	Struct.	Data base
	Ontology population	WEB->KB	Struc. & semi-struc.	Machine-learning
		ArtEquAKT	Unstruc.	NLP
	Semantic annotation	CREAM	Unstruc.	NLP and Data mining
		MnM	Unstruc.	NLP
KB extraction	Information integration	Ontominer	Semi-struc.	Hierarchical clustering

Cuadro 4.1: *Propuestas para la extracción de conocimiento.*

de minería de datos y de procesamiento de lenguaje natural para asistir al ingeniero del conocimiento en el ciclo de vida del desarrollo de ontologías.

ASIUM. Es un sistema cooperativo de aprendizaje automático capaz de obtener ontologías para dominios específicos de textos técnicos en lenguaje natural que han sido previamente sintácticamente tratados [22]. Está basado en un algoritmo no supervisado de clustering.

INTHELEX. Es un sistema incremental (empieza con una teoría vacía y la va completando) para la inducción de teorías jerarquizadas a partir de ejemplos [20]; de manera que la teoría aprendida se chequea para cada nuevo ejemplo y en caso de fallo se inicia una revisión del sistema.

El aprendizaje de esquemas trata con la extracción del conocimiento acerca de la estructura interna de las páginas web. Estos sistemas descubren reglas o patrones de conjuntos de páginas webs estructuradas y semi-estructuradas. El aprendizaje de esquemas a partir de páginas webs estructuradas [30, 72] se centra en la inferencia de DTDs, XML-Schemas o RDF-Schemas que describen un conjunto de documentos en XML. El aprendizaje de esquemas a partir de páginas webs semi-estructuradas [12, 70, 88, 89] se centra en descubrir árboles que usualmente aparecen en páginas web.

4.3. Sistemas de extracción de instancias

Los sistemas extractores de instancias intentan extraer las instancias de individuos de conceptos y las propiedades especificadas en una ontología externa. Se clasifican en: sistemas de respuesta automáticos, población de ontologías y sistemas de anotación semántica.

Los sistemas de respuesta automáticos permiten preguntar a la web como si fuera una base del conocimiento. Entre las propuestas más importantes se encuentran Squeal, y Omnibase y START.

Squeal. Muestra la web como si fuese una enorme base de datos, donde las relaciones estructurales son representadas como relaciones de base de datos [82]. Esta abstracción se materializa en una ontología llamada Squeal, que está especificada como un esquema SQL. El lenguaje de consultas SQL es usado para preguntar por el conocimiento acerca de la estructura de una página web y esto es posible ya que Squeal relaciona los elementos de HTML semánticamente.

Omnibase y START. START es un sistema de respuestas automáticas en lenguaje natural y Omnibase es una base de datos virtual que provee de un acceso uniforme a los recursos web [51]. START usa anotaciones procesables en lenguaje natural para describir las clases de las preguntas que se pueden contestar.

Los sistemas de población de ontologías tratan de alimentar ontologías externas con instancias extraídas de páginas web. Entre las propuestas más importantes se encuentran WEB→KB, y ArtEquAKT.

WEB→KB. Tiene como objetivo desarrollar una base de conocimiento probabilística que refleje el contenido de la web [15]. WEB→KB toma dos entradas: una ontología que define los conceptos y un conjunto de datos de entrenamiento compuesto de páginas webs etiquetadas con la información semántica que representa instancias de la ontología. Con estas entradas y aplicando varios algoritmos de aprendizaje automático aprende a extraer información de páginas similares y de los enlaces en esas páginas.

ArtEquAKT. Tiene como objetivo extraer automáticamente instancias de individuos acerca de artistas de la web para poblar una ontología [1]. Los individuos y la ontología forman una base del conocimiento que se utiliza para generar dinámicamente presentaciones personalizadas hechas a la medida de las necesidades de los usuarios (bibliografías narrativas).

Por último, los sistemas de anotación semántica tratan con la anotación de páginas web con meta-datos que describen la semántica del contenido en esas páginas web. Entre las propuestas más importantes se encuentran CREAM, y MnM.

CREAM. Es un banco de trabajo para definir metadatos usando ontologías [37, 38]. El banco de trabajo está compuesto de un gestor de documentos, editores de documentos, visores de documentos, una meta-ontología y un rasonador. CREAM soporta la anotación semi-automática de páginas web. Está basado en la herramienta de extracción de información Amilcare, que es capaz de realizar extracción single-slot de páginas web sin estructura. CREAM anota las páginas web usando reglas de extracción de conocimiento. Estas reglas son aprendidas a partir de un conjunto de páginas web ya anotadas.

MnM. Es una herramienta automática/semi-automática de anotación. Integra un navegador web, un editor de ontologías y una herramienta para la extracción de información, en un banco de trabajo que guía al usuario humano en el proceso de anotación semántica [86]. MnM define un proceso compuesto por cinco actividades: navegar, anotar, aprender, comprobar y extraer. En navegación, se accede a un servidor de ontologías con objeto de seleccionar un conjunto de modelos de interés para el sitio que se quiere anotar. La anotación tiene como objetivo que el usuario anote manualmente un conjunto de páginas de prueba. En aprendizaje, un algoritmo se ejecuta sobre dicho conjunto de páginas y generaliza reglas de extracción. En pruebas, se comprueba la precisión de las reglas utilizando páginas de prueba. Por último, en extracción, un algoritmo de extracción se ejecuta sobre las páginas web.

4.4. Sistemas de extracción de bases de conocimiento

Sólo existe una propuesta en la literatura que tiene como objetivo la extracción de ontologías e instancias de páginas webs. OntoMiner [16] usa técnicas de minería de datos para construir y poblar ontologías en un determinado dominio. Esta tarea la lleva a cabo organizando y minando un conjunto de sitios webs proporcionados por el usuario. Los autores lo presentan como un sistema de integración de información que transforma sitios web sintácticos a sitios web semánticos que permitien razonamiento automatizado.

Ontominer toma como entrada la URL de 10 a 15 sitios web y dirigidos por una taxonomía (una taxonomía organiza sus contenidos) que caracteriza al dominio de interés y usa varios algoritmos que detectan y usan las regularidades del código HTML de las páginas web para identificar relaciones jerárquicas entre los conceptos más importantes del dominio. A continuación, Ontominer expande la taxonomía de conceptos minada con subconceptos, para ello selectivamente visita los enlaces correspondiente a los conceptos ya identificados. Por último, Ontominer extrae instancias de las páginas web detectando los segmentos de interés usando para ello la jerarquía previamente obtenida.

Parte III

Nuestra aportación

Capítulo 5

Motivación

Ability is what you're capable of doing.
Motivation determines what you do.
Attitude determines how well you do it.

Lou Holtz, 1937–
American football coach

Las soluciones actuales para la extracción de información y de conocimiento de la web no son lo suficientemente prácticas ya que no son capaces de resolver todos los problemas que ocasiona el tratar con una web sintáctica, dinámica, masiva y distribuida. Este Capitulo está organizado de la siguiente manera: la Sección §5.1 introduce el capítulo; en la Sección §5.2, presentamos los problemas anteriores en detalle; por último, en la Sección §5.3, probamos que las soluciones actuales no resuelven todos los problemas anteriores.

5.1. Introducción

En los últimos años, el diseño de arquitecturas de referencia para sistemas distribuidos en Internet ha atraído a un numero importante de investigadores que han enfocado su trabajo en el diseño de plataformas, lenguajes, middlewares, o temas relacionados con la interoperabilidad. El principal motivo de este interés es que Internet se ha convertido en un entorno perfecto para realizar transacciones comerciales [49, 79, 85].

Un desafío importante para los participantes en las transacciones comerciales ha sido examinar una gran cantidad de información para encontrar productos y servicios útiles. Afortunadamente, la tecnología ha evolucionado e Internet ha madurado hasta un punto en el que han aparecido sofisticados agentes de nueva generación. Estos agentes permiten eficientes búsquedas en el vasto repositorio de información que nos ofrece la web, solucionando algunos problemas relacionados con el lento acceso a Internet y las prohibitivas esperas, al trabajar en segundo plano. Con objeto de que dichos agentes sean seguros e interoperables, deben ser capaces de conseguir acceso al conocimiento que reside en la web, lo que es difícil ya que la web actual es sintáctica, dinámica, masiva y distribuida.

La web semántica traerá significando a la web actual, haciendo posible que los agentes software puedan entender la información que contiene. Sin embargo, las tendencias actuales parecen sugerir que no es probable su adopción en un futuro inmediato. Por lo tanto, la extracción de información con significado por agentes software es un problema sin solucionar. Algunos autores trabajan en propuestas que tienen como objetivo la extracción de información o conocimiento que reside en la web, con objeto de permitir su recuperación y manipulación por aplicaciones software. Desgraciadamente, ninguna solución parece ser apropiada en el contexto de la ingeniería de software. La razón es que no solucionan todos los problemas citados anteriormente.

5.2. Problemas

Tres problemas hacen difícil la tarea de desarrollar agentes software que saquen provecho de la información que reside en la web:

La web es sintáctica. Dificultando que los agentes software entiendan la información que reside en ella. Los agentes software hacen uso de ontologías que modelan determinados dominios para comunicarse entre

ellos, y desarrollar complicadas tareas para sus usuarios. La web actual no hace uso de ontologías para describir semánticamente la información que proporciona, lo que impide que los agentes software puedan razonar automáticamente sobre el conocimiento en la web.

Teniendo en cuenta esta observación, parece claro que sería deseable para una solución que posibilitase que los agentes entendiesen la información residente en la web. De tal manera que un gente que use una ontología que especifica la información semántica de un sitio web, tenga acceso a un conjunto de instancias de individuos que relacionen la información en el sitio web con los conceptos definidos en la ontología.

La web cambia continuamente. La web es dinámica, evoluciona de forma constante e impredecible, dificultando a los ingenieros software desarrollar y mantener agentes software que explotan la web como un repositorio de información. Los agentes usan esa información para cumplir sus objetivos y si llegase a invalidarse, podría provocar fallos en su comportamiento con efectos inesperados.

Debido a que los cambios en la web no se pueden evitar, es deseable para una solución práctica tenerlos en cuenta y minimizar los costes de mantenimiento cuando se produzcan.

La web es enorme y distribuida. Conecta un número enorme de fuentes heterogéneas de información, y por tanto, un número inmenso de sitios web a los que podría acceder un agente para extraer conocimiento. El tratamiento manual se hace inviable y es necesario de soluciones automáticas y distribuidas para sacar provecho de todo este potencial.

Las características anteriores son necesarias para dar soporte fiable a los ingenieros software en el desarrollo de agentes que necesitan entender la información en la web a un coste razonable. Con un coste razonable nos referimos a encontrar un equilibrio entre razonamiento automático, adaptabilidad, automatización y distribución que minimice los costes de desarrollo y mantenimiento de agentes software.

5.3. Análisis de las soluciones actuales

Nuestro objetivo en esta sección es probar que ninguno de los sistemas presentados en los Capítulos §3 y §4 solucionan todos los problemas identificados en la sección anterior.

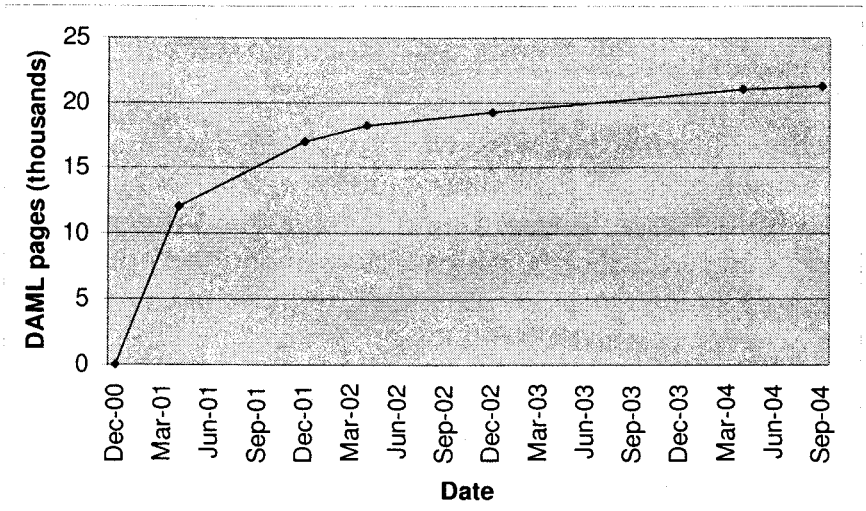


Figura 5.1: *Evolución de la web semántica.*

5.3.1. La web semántica

La web semántica simplificará y mejorará el proceso de extracción de información con significado, limitándolo a la existencia de lenguajes de consultas para fuentes de información anotadas con semántica. No obstante, la visión de la web semántica requiere mucho esfuerzo, ya que implica anotar las páginas web existentes. La Figura §5.1 muestra la evolución de la web semántica; sugiere que existen muy pocas páginas web anotadas si las comparamos con las número de páginas web no anotadas (aproximadamente 2.5 billones de páginas web, según estimaciones de Cyveillance). La tendencia actual sugiere que el la web semántica no será adoptada en los próximos años [41].

Por tanto, hoy en día es no realista asumir que los agentes software puedan entender la web. Justificando el trabajo que están realizando algunos autores en el desarrollo de nuevas propuestas para la extracción de conocimiento, y en la mejora de las ya existentes. Estas propuestas serán válidas mientras la web semántica no esté instaurada.

5.3.2. Wrappers inductivos

Los wrappers inductivos son perfectos para la extracción de información de la web a un coste razonable; pueden ser combinados con verificadores sintácticos para hacer frente a cambios en la web; y pueden incluso incorporar

algoritmos de re-inducción para recuperarse de errores sin intervención humana. No obstante, los wrappers inductivos no asocian semántica a la información extraída, siendo este su mayor inconveniente. Por lo tanto, la única forma de utilizar la información extraída en aplicaciones es incrustando todo el conocimiento sobre ella en la lógica funcional de esas aplicaciones.

5.3.3. Soluciones ad-hoc

Las soluciones ad-hoc incrustan el conocimiento necesario para traducir la información sintáctica extraída a información con significado en la lógica funcional de las aplicaciones. El problema que aparece con estas soluciones es debido al dinamismo inherente a los sitios web, ya que cambios en la estructura de la página web o en la información a extraer pueden invalidarlas, incrementando el coste de su mantenimiento.

5.3.4. Extractores de conocimiento

Como mencionamos en el Capítulo §2, ni las propuestas en aprendizaje de esquemas ni las de respuestas automáticas están relacionadas con nuestro trabajo. Los sistemas de extracción de ontologías tratan con un problema muy complicado. La extracción de conceptos y propiedades es un problema abierto que necesita de mucha investigación. La precisión de estos sistemas no es la suficiente como para que aplicaciones software hagan uso de sus resultados. Además, con objeto de incrementar la precisión, reducen su ámbito de aplicación tratando con dominios concretos y con un determinado tipo de páginas web (sin estructura). Estas ideas justifican que en nuestro marco de trabajo, un ingeniero del conocimiento sea el encargado de describir la información semántica sobre la información en la web.

Nuestro trabajo está más cercano a aquellas propuestas en el campo de la población de ontologías, anotación semántica y extracción de bases de conocimiento. No obstante, nuestro enfoque se diferencia claramente. Nuestro objetivo es el desarrollo de aplicaciones, es decir, la ingeniería del software.

Población de ontologías. Alimentar ontologías externas con la información extraída de páginas web es la aproximación tomada por los sistemas WEB→KB [15] y ArtEquAKT [1].

En WEB→KB, las reglas aprendidas en la fase de aprendizaje pueden ser usadas en otros sitios web en un mismo dominio de aplicación, lo que hace

que esta solución sea capaz de adaptarse a cambios en la web. Automatiza el proceso de extracción de instancias de la web, pero es necesario una compleja etapa de entrenamiento, lo que hace excesivo el coste a pagar para conseguirla. WEB→KB usa el wrapper SRV para la extracción de información sintáctica, lo que lo limita a realizar una extracción single-slot en páginas web estructuradas y semi-estructuradas. Además, la sincronización del conocimiento en la web con el de la base de conocimiento, hace no factible su uso en sitios web en los que la información cambia continuamente, por ejemplo, los valores de mercado.

ArtEquAKT trabaja en un domino concreto (biografías de artistas) y sobre páginas web sin estructura, lo que reduce su ámbito de aplicación y hace no escalable sus resultados a otros dominios. Además, no tiene ninguna estrategia para tratar los cambios en la web.

Anotación semántica. Estos sistemas tienen como objetivo acelerar la instauración de la web semántica. CREAM [37, 38] y MnM [86] son sistemas similares, ambos proveen a los usuarios de un banco de trabajo para anotar páginas web con meta-datos. En sus últimas versiones incorporan wrappers inductivos con objeto de anotar semi-automáticamente. El primer inconveniente de estos sistemas es que no dan solución para los cambios en la web, además, se enfocan a páginas web sin estructura lo que reduce su ámbito de aplicación. Por último, son soluciones desde el punto de vista del proveedor y no del consumidor de la información.

Extracción de bases de conocimiento. Ontominer [16] es una solución del campo de la integración de información. Su objetivo es convertir sitos web sintácticos en sitios web semánticos. Ontominer extrae automáticamente una taxonomía de sitios web que están dirigidos por alguna taxonomía; por tanto presenta los mismos problemas que los sistemas de aprendizaje de ontologías. Al igual que los sistemas de anotación semántica, es una solución desde el punto de vista del proveedor. Con respecto al proceso de extracción de instancias de individuos, viene guiado por la taxonomía obtenida, impidiendo la posibilidad de reutilizar ontologías existentes, y de decidir el conocimiento de interés de todo el que ofrece el sitio web. Tampoco tiene en cuenta los cambios en la web.

5.4. Discusión

De la sección anterior se deduce que un marco de trabajo capaz de extraer conocimiento, que asimile los cambios en la web, y que automatice el desarrollo de extractores de conocimiento distribuidos es deseable desde un punto de vista práctico.

WebMeaning es nuestra propuesta para la extracción de información con significado de la web actual no semántica. Da soporte de ingeniería para la construcción de agentes software, a un coste razonable, que usan el conocimiento residente en la web para satisfacer sus objetivos. Los extractores de conocimiento construidos con WebMeaning se llaman wrappers semánticos y están basados en una división de la funcionalidad necesaria en el proceso de extracción de conocimiento. Un wrapper semántico está compuesto por un wrapper sintáctico para la extracción de información sintáctica de la web, un verificador sintáctico para verificar la información extraída con objeto de detectar cambios en la web, un traductor semántico para dar semántica a la información extraída y un verificador semántico para chequear semánticamente el conocimiento. Estos elementos pueden ser libremente integrados para producir wrappers semánticos, de esta manera, se facilita su sustitución, reduciendo el impacto de los cambios y mejorando la reutilización.

WebMeaning permite incorporar cualquier wrapper inductivo o verificador sintáctico existente para la extracción y verificación de información, de esta forma obtiene el beneficio de todo el trabajo desempeñado en esos campos de investigación. Nuestra propuesta para la construcción de traductores semánticos se formaliza en el Capítulo §7 y una posible materialización en el Capítulo §8 donde se detallan algoritmos que guían el desarrollo semi-automático de traductores semánticos independientes del dominio. Por último, el uso de razonadores nos permite chequear la consistencia de la fuente de información que alimenta al sitio web.

Capítulo 6

El marco de trabajo WebMeaning

If you have knowledge, let others light their candles at it.

Margaret Fuller, 1810–1850
American transcendentalist author and editor

Nuestro objetivo en este capítulo es presentar el marco de trabajo que hemos diseñado para tratar el problema del desarrollo de agentes software capaces de entender los contenidos de la web sintáctica actual. Está organizado como sigue: la Sección §6.1 introduce informalmente el marco de trabajo a partir de un diagrama de actividades; la Sección §6.2 presenta algunos conceptos preliminares; finalmente, la Sección §6.3 define de forma rigurosa los elementos del marco de trabajo.

6.1. Introducción

Tal y como dijimos en el capítulo anterior, nuestro trabajo de investigación se centra en dar soporte de ingeniería a los desarrolladores de agentes software que necesitan consumir el conocimiento residente en la web para satisfacer sus objetivos. Dicho soporte se materializa en un marco de trabajo abstracto al que llamamos WebMeaning, que provee los mecanismos necesarios para la construcción de extractores de conocimientos a los que llamamos wrappers semánticos. Los wrappers semánticos son diseñados aplicando el principio de separación de conceptos. El criterio usado es hacer cada tarea del proceso de extracción de conocimiento una actividad, que es desarrollada por un determinado elemento; el wrapper semántico será el encargado de orquestar esos elementos para alcanzar el objetivo de extraer conocimiento.

La Figura §6.1 hace uso de la notación SPEM basada en UML, para mostrar intuitivamente las actividades involucradas en la extracción de conocimiento de la web: extracción de información, verificación de la información, traducción de la información a conocimiento y verificación del conocimiento. Estas actividades son realizadas por un wrapper sintáctico, un verificador sintáctico, un traductor semántico y un verificador semántico, respectivamente. El wrapper semántico orquesta todos esos elementos como sigue: dada una página web, invoca al wrapper sintáctico para extraer la información de interés de la página web; a continuación, el verificador sintácico se encarga de chequear su validez desde el punto de vista sintáctico; si la información es inválida, informa de un error sintáctico; en caso contrario, el traductor semántico convierte la información a instancias de una ontología. De nuevo, el conocimiento es chequeado para detectar errores semánticos; si el conocimiento no es válido, el verificador semántico informará de un error semántico, en caso contrario, todo ha ido bien y se devuelve las instancias.

6.2. Preliminares

A continuación definimos cuatro tipos de datos que especifican las entradas y salidas de los elementos de la Figura §6.1. Estos tipos son: página web, información estructurada (formato de la salida de los wrappers sintácticos), Abox (formato de la salida de los traductores semánticos) y el resultado del proceso de extracción de conocimiento.

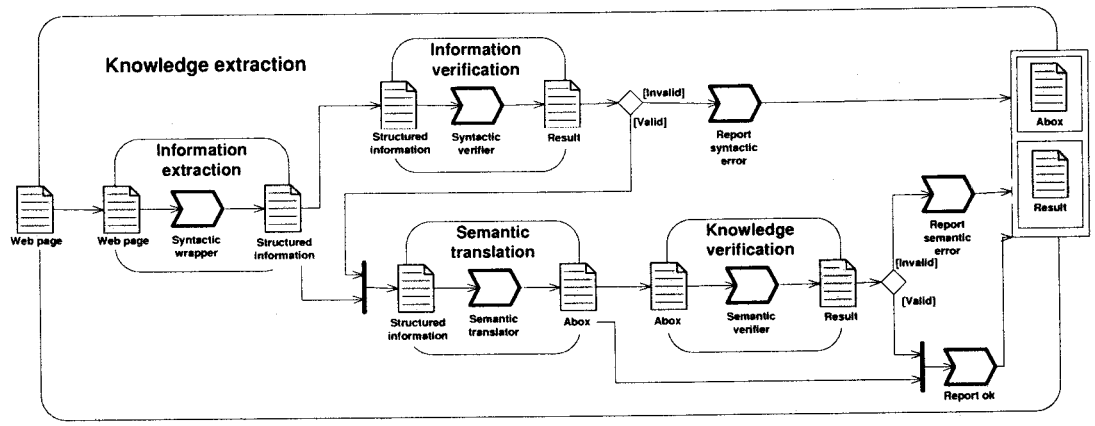


Figura 6.1: *Flujo de trabajo de un traductor semántico.*

6.2.1. Páginas web

Las páginas web son modeladas introduciendo un tipo básico llamado *WebPage*, que representa al conjunto de todas las páginas webs existentes.

[*WebPage*]

6.2.2. Salida de los wrappers sintácticos

Definimos el tipo de datos *StructuredInformation* para representar la salida de los wrappers. Está compuesto por un conjunto de *HierarchicalSlots*, lo que nos permite trabajar con wrappers capaces de realizar una extracción multi-slot. El tipo de datos *HierarchicalSlot* está compuesto de un árbol t y un par de funciones: *vertexAttributes* y *vertexHLevel*, lo que nos permite representar información jerárquica. El tipo abstracto de datos *Tree* está definido en el apéndice §A y especifica un árbol con raíz, como un grafo conectado y acíclico compuesto de un conjunto de vértices y ejes. La Función *vertexHLevel* asocia ejes a valores de tipo *Label* (que representa al conjunto de todas las etiquetas). Cada etiqueta representa un nivel jerárquico. La función *vertexAttributes* asocia vértices a seq *Attribute*, representando un trozo de información proveida en un nivel jerárquico y permitiéndonos establecer una posición para cada atributo (resolviendo el problema de representar permutaciones de los atributos). *Attribute* representa a un atributo a extraer y viene especificado como un conjunto de literales; lo que nos permite tratar el problema de los atributos opcionales y atributos multivaluados. Además, cuatro predicados formalizan los

requisitos que debe de cumplir un *StructuredInformation*: el primero y el segundo garantizan que las funciones *vertexAttributes* y *vertexHLevel* están definidas para todos los vértices en *t*; el tercero asegura que todos los slots en un mismo nivel jerárquico se encuentran a la misma profundidad en el árbol y que todos tienen el mismo número de atributos; el último garantiza que todos los vértices tengan al menos un atributo.

$[Literal, Label]$
$StructuredInformation == \mathbb{P}\, HierarchicalSlot$
$Attribute == \mathbb{P}\, Literal$

HierarchicalSlot

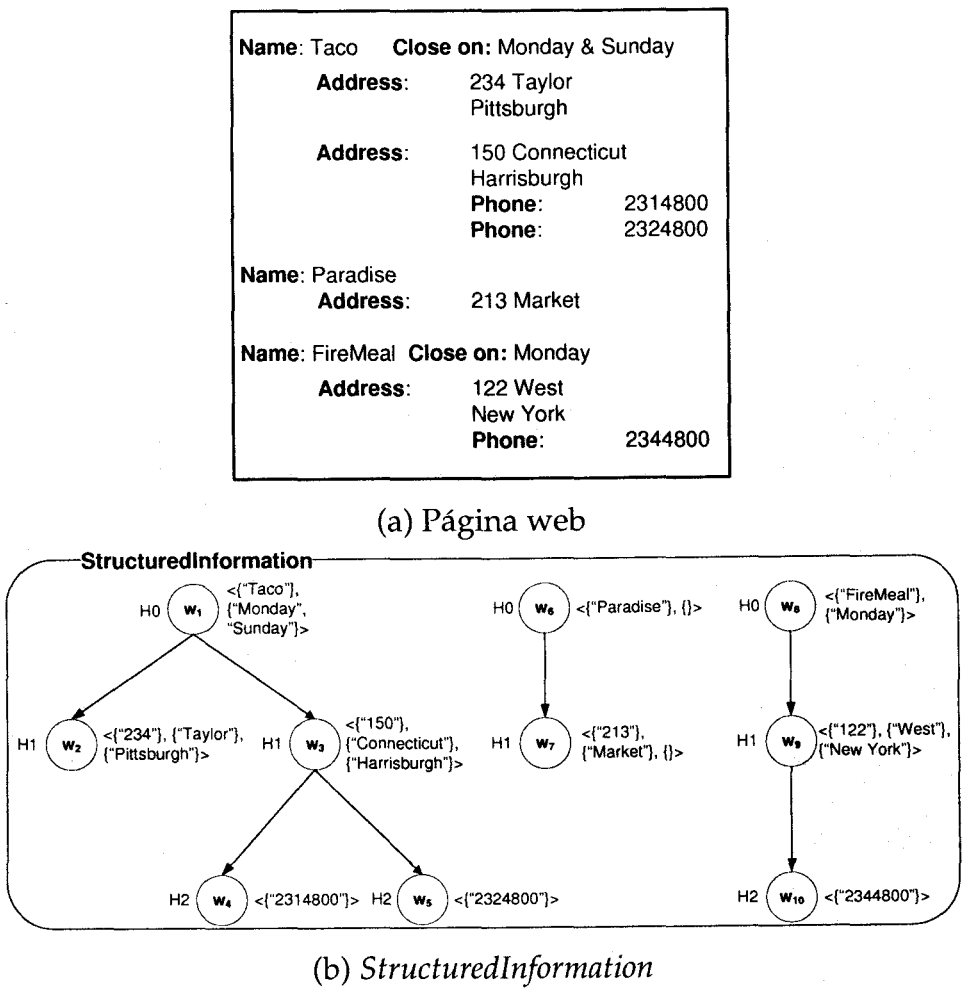
$t : Tree$
$vertexAttributes : Vertex \pfun \operatorname{seq} Attribute$
$vertexHLevel : Vertex \pfun Label$

$\operatorname{dom} vertexAttributes = t.vertices$
$\operatorname{dom} vertexHLevel = t.vertices$
$\forall w_1, w_2 : Vertex \mid w_1 \neq w_2 \land vertexHLevel(w_1) = vertexHLevel(w_2) \bullet$
$\quad \#vertexAttributes(w_1) = \#vertexAttributes(w_2) \land sameDepth(t, w_1, w_2)$
$\forall w : \operatorname{dom} vertexAttributes \bullet$
$\quad \#vertexAttributes(w) > 0$

Ejemplo 6.1 *La Figura §6.2(b) muestra la StructuredInformation extraída para la página web de la Figura §6.2(a). En ella aparecen tres niveles jerárqicos: H0, que representa informción acerca de un restaurante (nombre y los días que cierra); H1 que representa la información sobre direcciones (número, calle y ciudad); y H2, que encapsula la información acerca de números de teléfonos (número). Cada slot jerárquico representa información sobre un restaurante, sus direcciones y números de teléfonos asociados a las direcciones; Por ejemplo, el primer slot jerárquico representa un restaurante con dos direcciones, donde la segunda dirección tiene asociada dos números de teléfono; el segundo, representa un restaurante con una única dirección y sin números de teléfono.*

6.2.3. Aserciones sobre individuos

WebMeanin utiliza lógica descriptiva para representar instancias de individuos. Tal y como mencionamos en la Sección§2, un Abox es un conjunto



(a) Página web

(b) *StructuredInformation*

Figura 6.2: *Ejemplo de StructuredInformation.*

de aserciones de axiomas. Existen dos clases de aserciones: aserciones de conceptos y aserciones de propiedades. Una aserción de un concepto declara que un individuo es la instancia de un concepto; una aserción de una propiedad declara que un individuo está relacionado con otro individuo (relación entre dos instancias de conceptos) o con un literal (atributo asociado a un concepto).

El esquema abajo formaliza un Abox. La parte declarativa está compuesta por seis conjuntos: nombres de conceptos, nombres de propiedades, nombres de individuos, aserciones de conceptos y aserciones de propiedades. La parte de predicados está vacía, esto es debido a que delegamos la verificación del Abox a razonadores.

$[ConceptName, PropertyName, IndividualName]$

Abox

$$
\begin{array}{l}
conceptNames : \mathbb{P}\, ConceptName \\
propertyNames : \mathbb{P}\, PropertyName \\
individualNames : \mathbb{P}\, IndividualName \\
conceptAssertions : \mathbb{P}(conceptNames \times individualNames) \\
propertyAssertions : \mathbb{P}(propertyNames \times individualNames \times (individualNames \cup Literal))
\end{array}
$$

Ejemplo 6.2 *La Figura §6.3 muestra el Tbox para casas de comida. Usando este Tbox, definimos el Abox en nuestro caso de estudio como sigue:*

$$
\begin{aligned}
a = (\!| \; & \textbf{\textit{conceptNames}} \rightsquigarrow \{Restaurant, Address, City, Phone\}, \\
& \textbf{\textit{propertyNames}} \rightsquigarrow \{name, closed, hasAddress, street, number, inCity, hasPhone\}, \\
& \textbf{\textit{individualNames}} \rightsquigarrow \{RID_1, AID_1, AID_2, CID_1, CID_2, PID_1, PID_2\}, \\
& \textbf{\textit{ConceptAssertions}} \rightsquigarrow \{(Restaurant, RID_1), (Address, AID_1), \\
& \quad (Address, AID_2), (City, CID_1), (City, CID_2), (Phone, PID_1), \\
& \quad (Phone, PID_2)\} \\
& \textbf{\textit{propertyAssertions}} \rightsquigarrow \{(name, RID_1, \text{"Taco"}), (closed, RID_1, \text{"Monday"}), \\
& \quad (closed, RID_1, \text{"Sunday"}), (hasAddress, RID_1, AID_1), (street, AID_1, \text{"Taylor"}), \\
& \quad (number, AID_1, \text{"234"}), (inCity, AID_1, CID_1), (name, CID_1, \text{"Pittsburgh"}), \\
& \quad (hasAddress, RID_1, AID_2), (street, AID_2, \text{"Connecticut"}), (number, AID_2, \text{"150"}), \\
& \quad (inCity, AID_2, CID_2), (name, CID_2, \text{"Harrisburgh"}), (hasPhone, AID_2, PID_1), \\
& \quad (number, PID_1, \text{"2314800"}), (hasPhone, AID_2, PID_2), (number, PID_2, \text{"2324800"})\} \; |\!)
\end{aligned}
$$

Este Abox, usa el vocabulario de a.conceptNames ∪ a.propertyNames para declarar aserciones sobre un individuo del concepto Restaurant. Por ejemplo, la aserción de concepto $(Restaurant, RID_1)$ *significa que* RID_1 *es una instancia del concepto Restaurant;*

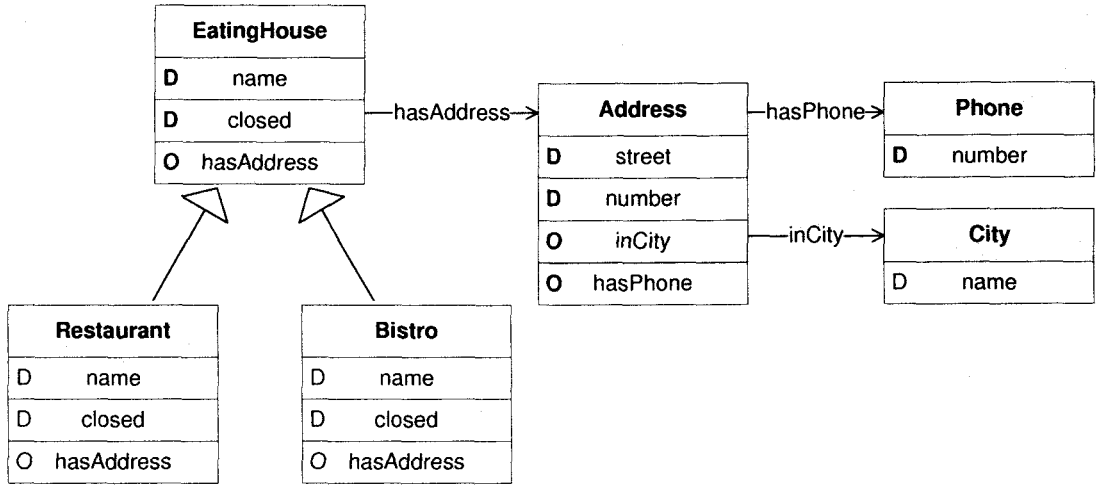


Figura 6.3: *Una ontología acerca de casas de comida.*

la aserción de propiedad $(hasAddress, RID_1, AID_1)$ *significa que* AID_1 *es una dirección del restaurante* RID_1 *y* $(name, RID_1, "Taco")$ *significa que el valor de la propiedad name para* RID_1 *es "Taco".*

Por último, y de acuerdo con la lógica descriptiva, una base de conocimiento se modela como el producto Cartesiano de los tipos *Tbox* y *Abox*. Observe que introducir *Tbox* como un tipo libre nos permite una máxima libertad en la elección de una lógica descriptiva concreta.

$[Tbox]$
$KnowledgeBase == Tbox \times Abox$

6.2.4. Resultado del proceso

El resultado del proceso de extracción de conocimiento es un tipo enumerado. Introducimos *Result* como el conjunto más pequeño que contiene las tres constantes: *OK*, *SYNTFAIL* y *SEMFAIL*. *OK* significa que no se han producido fallos en el proceso. *SYNTFAIL* y *SEMFAIL*, informan de fallos sintácticos y semánticos, respectivamente.

$Result ::= OK \mid SYNTFAIL \mid SEMFAIL$

6.3. Definiciones nucleares

El núcleo de nuestro marco de trabajo es un conjunto de definiciones que rigurosamente especifican los elementos presentados en la Figura §6.1.

6.3.1. Wrappers sintácticos

Definición 6.1 (Wrappers sintácticos) *Un wrapper sintáctico es una función que toma una página web a la entrada y devuelve una vista estructurada de la información de interés.*

La función axiomática definida abajo formaliza un wrapper sintáctico. Está definida como una función parcial porque su dominio el el coonjunto de páginas webs que definen su alcance (las páginas web en las que podemos usar el wrapper).

$$Wrapper : WebPage \pfun StructuredInformation$$
$$\dom Wrapper \neq \varnothing$$

En WebMeaning se pueden usa cualquier wrapper inductivo, wrappers desarrollados manualmente o APIs proporcionadas por los sitios web. La implementación de wrappers inductivos permite automatizar el proceso de desarrollo de wrappers. Los wrappers desarrollados manualmente se desarrollan rápidamente, no obstante los costes de mantenimiento pueden ser altos; sin embargo esta estrategia puede ser interesante si los cambios en las páginas no son muy frecuentes o están controlados. Por último, algunos sitios proveen APIs para acceder a la información, por ejemplo, `Google.com` para resultados de búsquedas, `Amazon.com` para información de libros o `Imdb.com` para información de películas. Estas APIs no son siempre gratis, sin embargo el coste es razonable si consideramos que la información que devuelven no se ve afectada por cambios en la estructura del sitio.

A la hora de seleccionar la implementación de un wrapper inductivo o por el contrario implementarlo manualmente, los desarrolladores deberían hacerse algunas preguntas sencillas acerca del sitio web en el que reside la información y como la información está presentada:

- ¿Es necesario extracción single-slot o multi-slot?

- ¿Son las páginas web estructuradas, semi-estructuradas o sin estructura?
- ¿Contiene la página web atributos opcionales, o atributos multivaluados o permutaciones en los atributos?
- ¿Está la información en la página web estructurada jerárquicamente o en forma de tabla?

6.3.2. Verificadores sintácticos

Definición 6.2 (Verificadores sintácticos) *Un verificador sintáctico es un predicado que toma la información extraída por un wrapper y se satisface si el wrapper está trabajando correctamente. Se llaman verificadores sintácticos porque la decisión acerca de la validez del wrapper está basada únicamente en propiedades sintácticas de la información extraída.*

$$| SyntacticVerifier : StructuredInformation$$

Por lo tanto, un wrapper W es fiable si satisface la siguiente fórmula:

$$\forall p : WebPage \mid p \in \mathrm{dom}\, W \bullet SyntacticVerifier(W(p))$$

WebMeaning tiene en cuenta el problema de los cambios en la web definiendo los verificadores sintácticos. En WebMeaning se puede usar cualquier verificador ya existente, como RAPTURE o DataProG, o verificadores creados manualmente. Al igual que los sistemas de wrappers inductivos, la implementación de RAPTURE o DataProG permiten automatizar el desarrollo de verificadores sintácticos. Si se opta por el desarrollo manual, un resultado de Kushmerick [54] puede resultar interesante: tan solo teniendo en cuenta la densidad HTML (fracción de '<' y '>' en la información extraída) se identifican correctamente casi todos los cambios. Por último, si estamos usando una API proveida por un sitio web, no es necesario implementar un verificador sintáctico (el predicado *SyntacticVerifier* se satisface para todas las páginas web del sitio).

6.3.3. Traductores semánticos

Definición 6.3 (Semantic translators) *Un traductor semántico es una función que toma a la entrada la información extraída de un sitio web (StructuredInformation) y*

devuelve el Abox semánticamente equivalente a dicha información. Para realizar esta tarea puede hacer uso de toda la información disponible, como instancias correctas de individuos, ontologías o reglas de mapeo definidas por los usuarios.

$$SemanticTranslator : WebPage \nrightarrow Abox$$
$$\text{dom } SemanticTranslator \neq \varnothing$$

Antes de implementar un traductor semántico es necesario determinar el lenguaje de especificación de ontologías que más se ajusta a nuestras necesidades. Podemos decidir apostar por una máxima expresividad, o por eficiencia computacional o equilibrar expresividad y eficiencia. Las ideas en el Capítulo §2, resumidas en las tablas §2.1 y §2.2 pueden ayudarnos en esta elección. Dependiendo del lenguaje usado tendremos las ventajas de usar un razonador; por ejemplo, si elegimos OWL-Lite, entonces podemos utilizar RACER como razonador, por el contrario, si elegimos OWL-Full, entonces tenemos que ser consientes de que no tendremos la posibilidad de razonar sobre el conocimiento extraído.

Los siguientes dos capítulos presentan nuestra propuesta para la construcción de traductores semánticos. El capítulo §7 los presenta de forma abstracta y en el Capítulo §8 se definen algoritmos concretos que guían su desarrollo.

6.3.4. Verificadores semánticos

Definición 6.4 (Verificadores semánticos) *Un verificador semántico chequea la satisfabilidad de un conjunto de instancias de una ontología. Es decir, chequea que las instancias sean consistentes con as restricciones semánticas impuestas en la ontología.*

Un verificador semántico está especificado como un predicado que chequea la base de conocimiento usando un razonador.

$$SemanticVerifier : KnowledgeBase$$

Observe que los verificadores semánticos detectan inconsistencias en la fuente de información usada para alimentar el sitio web, mientras que los verificadores sintácticos detectan cambios en la estructura del sitio web que invalidan el proceso de extracción. La solución a los errores semánticos no es

la reconstrucción del wrapper, dado que sigue trabajando bien, sino esperar a que la fuente de información sea correcta o buscar otro sitio que ofrezca la misma información.

Como hemos mencionado anteriormente, si no existe razonadores para el lenguaje ontológico usado, el conocimiento extraído no podrá ser chequeado (el predicado *SyntacticVerifier* se satisface para cualquier Abox).

6.3.5. Wrappers semánticos

Definición 6.5 (Wrappers semánticos) *Un wrapper semántico es una función que toma una página web a la entrada y devuelve un conjunto de instancias de conceptos, definidos en una ontología, que representa la información de interés.*

Abajo, especificamos un wrapper semántico como una función axiomática que rigurosamente presenta la colaboración entre elementos concretos: un wrapper sintáctico (W), un verificador sintáctico (V), un traductor semántico (ST) y un verificador semántico (SV). *NULLABOX* representa un Abox vacío. La colaboración fue presentada informalmente en la Sección §6.1.

$$NULLABOX == \langle\!| \, conceptNames = \varnothing, propertyNames = \varnothing, individualNames = \varnothing, \\ conceptAssertions = \varnothing, propertyAssertions = \varnothing \, |\!\rangle$$

$$SemanticWrapper : WebPage \pfun Abox \times Status$$

$$\begin{aligned}
&\forall p : WebPage; a : Abox; si : StructuredInformation \mid p \in \mathrm{dom}\, Wrapper \bullet \\
&\quad (SemanticWrapper(p) = (a, OK) \land \\
&\qquad si = W(p) \land V(si) \land a = ST(si) \land SV(a)) \\
&\quad \lor \\
&\quad (SemanticWrapper(p) = (NULLABOX, SYNTFAIL) \land \\
&\qquad si = W(p) \land \neg\, V(si)) \\
&\quad \lor \\
&\quad (SemanticWrapper(p) = (NULLABOX, SEMFAIL) \land \\
&\qquad si = W(p) \land V(si) \land a = ST(si) \land \neg\, SV(a))
\end{aligned}$$

Capítulo 7

Traducción semántica

Everywhere one seeks to produce meaning, to make the world signify, to render it visible. We are not, however, in danger of lacking meaning; quite the contrary, we are gorged with meaning and it is killing us.

Jean Baudrillard, 1929–
French semiologist

En este capítulo, presentamos nuestra propuesta para la traducción semántica. El capítulo está organizado como sigue: en la Sección §7.1, introducimos las ideas más importantes; la Sección §7.2 define el problema de la traducción semántica y resume nuestra solución; las Secciones §7.3, §7.4, §7.5, §7.6 y §7.7 presenta nuestra solución en detalle.

7.1. Introducción

En éste capítulo presentamos de manera abstracta nuestra solución para la traducción semántica. Nuestra solución involucra la extracción de las relaciones semánticas de la información a extraer y relacionar dichas relaciones semánticas con la información extraída por el wrapper. Con esos resultados configuramos un algoritmo independiente del dominio al que llamamos traductor semántico, que es capaz de dar semántica de forma automática a la información que extrae un wrapper. Por tanto, la solución al problema de la traducción semántica consiste en dar como entrada la información extraída por el wrapper a dicho algoritmo.

7.2. Definición del problema

Para dar solución al problema de la traducción semántica, haremos algunas asunciones sobre los atributos en un *HierarchicalSlot*, las relaciones entre distintos niveles jerárquicos y la manera en la que se traducen los *HierarchicalSlot*:

Asunción 1. Asumimos que cada atributo en un *HierarchicalSlot* es el valor de una propiedad asociada con un concepto. Es decir, el valor de una propiedad no se puede calcular a partir de uno o varios atributos.

Asunción 2. Cada eje entre dos vértices en un *HierarchicalSlot* representa una propiedad multivaluada que relaciona dos conceptos diferentes.

Asunción 3. La información a extraer es acerca de un único concepto, el cual no puede ser especializado.

Dada la definición de traductor semántico en el Capítulo §6 y las suposiciones anteriores, nuestra solución a la traducción semántica implica tres tareas: descubrir la semántica de la información extraída, relacionar la información extraída con la información semántica y configurar un algoritmo genérico e independiente del dominio. El diagrama de actividad de la figura §7.1 da una vista general de estas tareas.

7.3. Representación de individuos

La especificación del tipo de datos *Abox* presentada en el capítulo §6 es una especificación abstracta. En esta sección presentamos una especificación

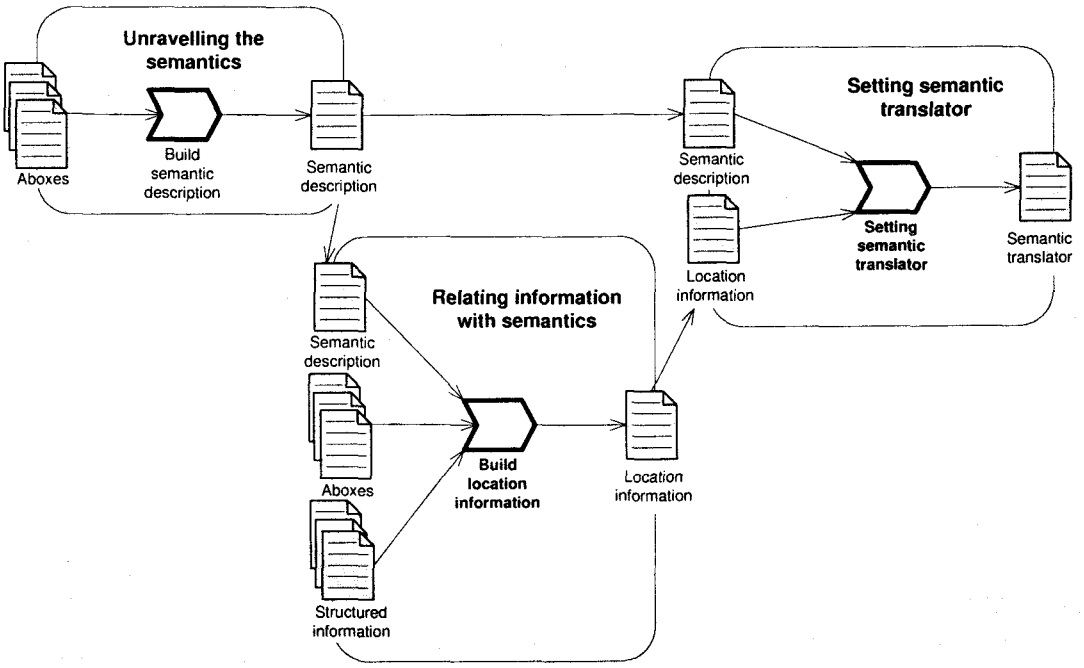


Figura 7.1: *Actividades para construir un traductor semántico.*

concreta de las aserciones sobre un único individuo en un Abox. Ambas especificaciones son equivalentes según se demuestra en el apéndice §B.

Definición 7.1 (Individual tree) *Un individual tree es la representación en forma de árbol de un individuo en un Abox. Existen dos tipos de vértices en el árbol: vértices que representan aserciones de conceptos, y vértices que representan atributos de conceptos. Los ejes representan propiedades y unen dos vértices, o entre un vértice instancia de concepto y otro vértice que representa un valor literal.*

Ejemplo 7.1 *La figura §7.2 muestra el individual tree identificado por* RID_1 *en el Abox del ejemplo §6.2.*

Con objeto de hacer más legible la especificación definiremos el tipo abstracto de datos *LabelledTree* que será muy usado en este capítulo. *LabelledTree* modela un árbol en el que los vértices son etiquetados con nombres de conceptos y los ejes con nombre de propiedades. Usamos el nombre de concepto *FILLER* para representar valores literales.

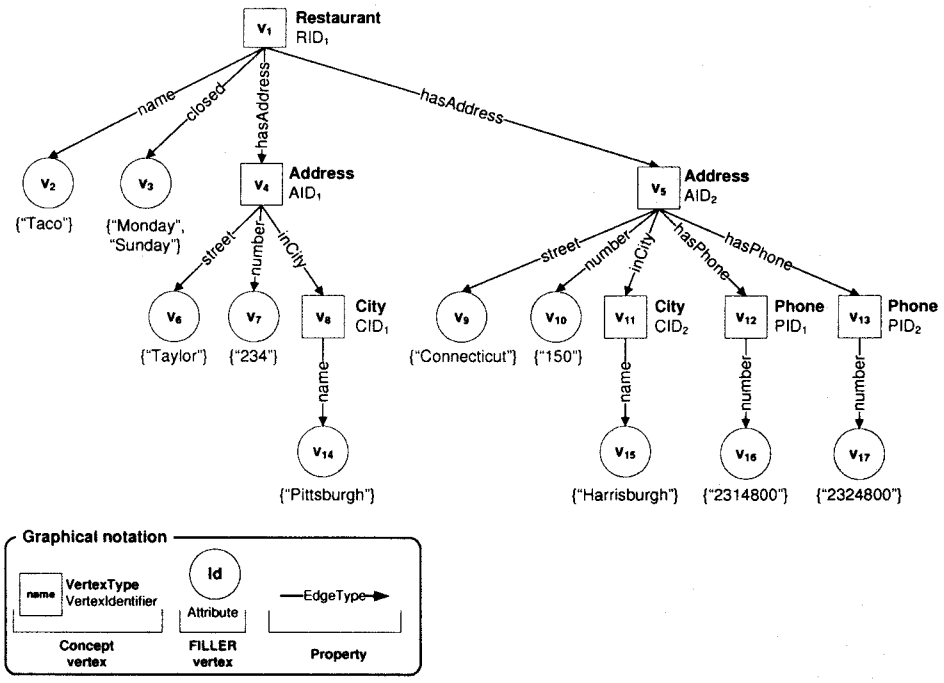


Figura 7.2: *Un individual tree.*

LabelledTree

$t : Tree$
$vertexType : Vertex \pfun ConceptName$
$edgeType : Edge \pfun PropertyName$

$\mathrm{dom}\, vertexType = t.vertices$
$\mathrm{dom}\, edgeType = t.edges$
$\forall (v_1, v_2) : t.edges \bullet vertexType(v_1) \neq FILLER$
$\forall v : t.vertices \mid isLeaf(t, v) \bullet vertexType(v) = FILLER$

Abajo, definimos el esquema *IndividualTree*, que incluye la variable *lt* de tipo *LabelledTree* y dos funciones que etiquetan los vértices: *vertexIndividualName* etiqueta los vértices con los nombres de individuos definidos en el Abox y *vertexAttribute* los etiqueta con un dato del tipo *Attribute*. Los predicados restringen el dominio de *vertexIndividualName* y *vertexAttribute* a vértices de tipo concepto y de tipo *FILLER*, respectivamente.

IndividualTree

$lt : LabelledTree$
$vertexIndividualName : Vertex \pfun IndividualName$
$vertexAttribute : Vertex \pfun Attribute$

$\dom vertexIndividualName = \{v : lt.t.vertices \mid lt.vertexType(v) \neq FILLER \bullet v\}$
$\dom vertexAttribute = \{v : lt.t.vertices \mid lt.vertexType(v) = FILLER \bullet v\}$

7.4. Descripciones semánticas

Las descripciones semánticas nos permiten representar en forma de árbol la semántica de la información extraída de la web. Por tanto, definen la semántica involucrada en los individual trees que representan a todos los individuos de un sitio web.

Definición 7.2 (Descripciones semánticas) *Una descripción semántica es una notación simple que representa en forma de árbol la semántica de la información extraída por un wraper. Define conceptos que representan entidades en un dominio, propiedades que describen la relación entre conceptos, o atributos, y restricciones de cardinalidad en las propiedades.*

Ejemplo 7.2 *La Figura §7.3 muestra la descripción semántica para nuestro caso de estudio. Su significado es el siguiente: Una instancia del concepto Restaurant tiene tres propiedades: name, closed y hasAddress. Las primeras dos propiedades son atributos. La propiedad hasAddress relaciona dos conceptos, un Restaurant con uno o más Address. El concepto Address viene definido por las propiedades street, number, inCity, y Phone. inCity relaciona un concepto Address con un concepto City (una dirección tiene asociada opcionalmente una ciudad). El concepto City tiene un Literal como name. Finalmente, un Phone tiene un Literal como number.*

El siguiente esquema especifica a una descripción semántica. Está compuesto por un árbol etiquetado y un conjunto de vértices (*repeated*) que define restricciones de cardinalidad en propiedades. Los predicados aseguran que solo los vértices de tipo concepto pueden estar en el conjunto *repeated* y que no existen vértices colapsables en el árbol etiquetado. El significado de vértices colapsables se formaliza en la Sección §7.5, intuitivamente, el predicado garantiza de que no existe más de un vértice representando la misma semántica.

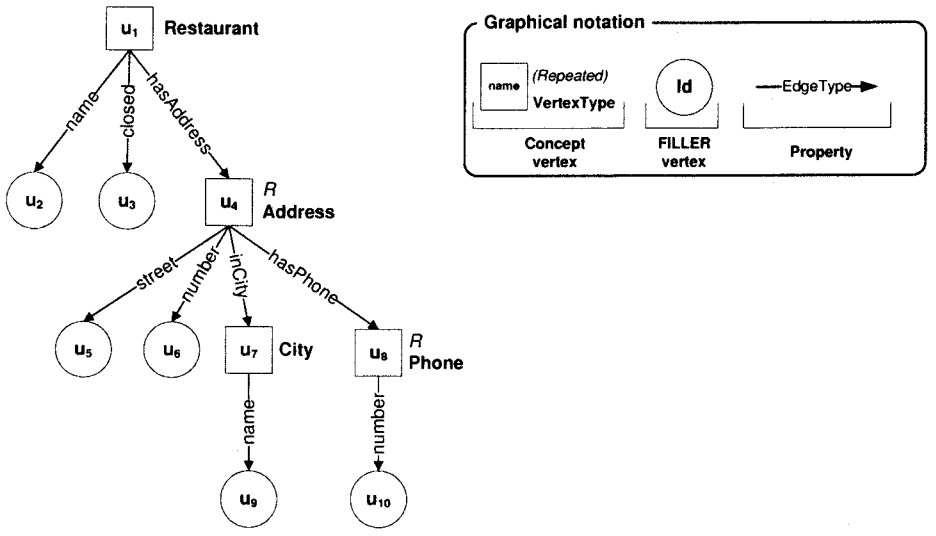


Figura 7.3: *Una descripción semántica.*

SemanticDescription

$lt : LabelledTree$

$repeated : \mathbb{P}\, vertices$

$\forall u : lt.t.vertices \mid lt.vertexType(u) = FILLER \bullet u \notin repeated$

$\forall su : \mathbb{P}\, lt.t.vertices \bullet \neg\, collapsableVertices(lt, su)$

7.4.1. Restricciones de cardinalidad

La Figura §7.4 ilustra graficamente el significado de las restricciones de cardinalidad en los eje de una descripción semántica. Los ejes pueden ser clasificados basándonos en el tipo de los vertices y si los vértices están en el conjunto *repeated* de la descripción semántica. La clasificación es la siguiente:

1. Un eje P entre dos vértices C_x y C_y de tipo concepto, en los que el vértice asociado a C_y está en el conjunto *repeated* de la descripción semántica, indica que P representa una relación múliple ($[0\,..\,n]$), en otras palabras, P relaciona una instancia de C_x con cero o más instancias de C_y.

2. Un eje P entre dos vértices C_x y C_y de tipo concepto que no están en el conjunto *repeated* de la descripción semántica, indica que P es una relación opcional ($[0\,..\,1]$), es decir, P opcionalmente relaciona una instancia de C_x con una instancia de C_y.

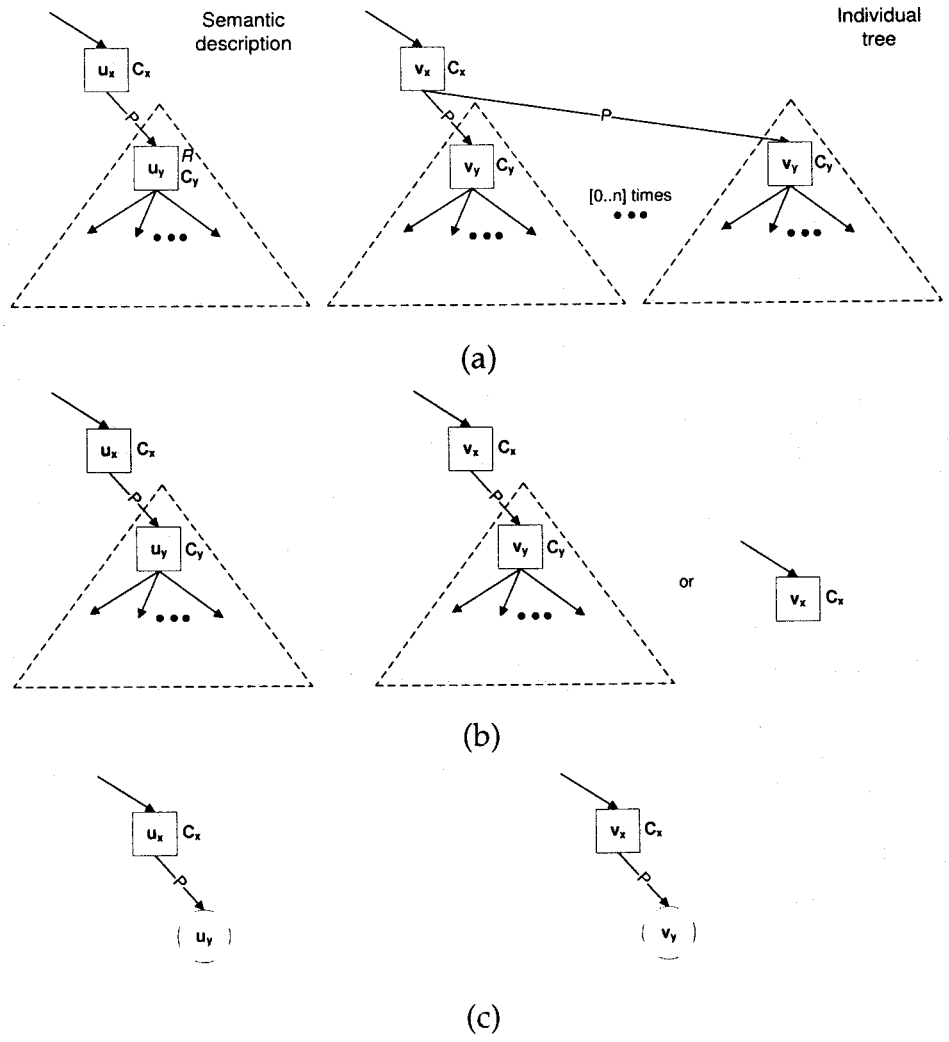


Figura 7.4: *Diferentes tipos de ejes en una descripción semántica.*

3. Un eje P entre un vértice C_x de tipo concepto y un vértice de tipo *FILLER*, indica que P es una relación múltiple ($[0 \,..\, n]$), es decir, P relaciona una instancia de C_x con cero o más literales.

7.4.2. Semántica

La semántica formal de una descripción semántica es especificada usando interpretaciones en lógica de primer orden [3]. Sea $\mathcal{L}$ un lenguaje lógico; una descripción semántica se interpreta como una tupla (P, A), donde P es un subconjunto del vocabulario de los símbolos de predicado de $\mathcal{L}$, y A es un subconjunto de formulas bien formadas en $\mathcal{L}$ (axiomas). Por tanto, una descripción semántica es interpretada como un subconjunto de $\mathcal{L}$ en el que los conceptos y propiedades son especificados como predicados unarios y binarios, respectivamente.

Dado *sd* : *SemanticDescription*, la traducción es como sigue:

- Cada nombre de concepto en la descripción semántica (ran *sd.lt.vertexType*\ {*FILLER*}) se interpreta como un símbolo de predicado unario. También, *Literal* se considera un nombre de concepto que representa a todas las cadenas.
- Cada nombre de propiedad en la descripción semántica (ran *sd.lt.edgeType*) es un símbolo de predicado binario.
- Cada nombre de propiedad añade un axioma que restringe el dominio y rango de esa propiedad:

  $$\forall x \bullet \exists y \bullet P(x,y) \Rightarrow \\ (C_{d_1}(x) \land (C_{r_1}(x)) \lor (C_{d_2}(x) \land (C_{r_2}(x)) \lor \ldots \lor (C_{d_n}(x) \land (C_{r_n}(x))$$

  donde P es el nombre de la propiedad; y las parejas (C_{d_i}, C_{r_i}) son elementos de un conjunto definido como:

  $$\{(v_1, v_2) : sd.lt.t.edges \mid sd.lt.edgeType(v_1, v_2) = P \bullet \\ (sd.lt.vertexType(v_1), sd.lt.vertexType(v_2))\}$$

- Cada eje (v_1, v_2) de tipo (ii) en la descripción semántica añade un axioma que restringe la cardinalidad de la propiedad es cero o una:

  $$\forall x, y, z \bullet P(x,y) \land P(x,z) \land C_1(x) \land C_2(y) \land C_2(z) \Rightarrow y = z$$

  donde

  $$P \triangleq sd.lt.edgeType(v_1, v_2), C_1 \triangleq sd.lt.vertexType(v_1), \text{ y } C_2 \triangleq sd.lt.vertexType(v_2)$$

Observe que los ejes de tipo (i) y (iii) no proporcionan información alguna sobre la cardinalidad de las propiedades, por lo tanto, puede existir más de una aserción de una propiedad para un mismo concepto.

Una interpretación de los conceptos y de las propiedades (definidos por los axiomas anteriores) sobre el dominio de la información extraída es un conjunto de aserciones predicados.

Ejemplo 7.3 *En nuestro ejemplo:*

$P = \{$***[From concept names]***
$Restaurant, Address, Phone, City, Literal$
[From property names]
$name, closed, hasAddress, street, number, inCity, hasPhone\}$
$A = \{$
[From property names]
$\forall x \bullet \exists y \bullet name(x,y) \Rightarrow (Restaurant(x) \wedge Literal(y)) \vee City(x) \wedge Literal(y))$
$\forall x \bullet \exists y \bullet closed(x,y) \Rightarrow Restaurant(x) \wedge Literal(y)$
$\forall x \bullet \exists y \bullet hasAddress(x,y) \Rightarrow Restaurant(x) \wedge Address(y)$
$\forall x \bullet \exists y \bullet street(x,y) \Rightarrow Restaurant(x) \wedge Literal(y)$
$\forall x \bullet \exists y \bullet number(x,y) \Rightarrow (Address(x) \wedge Literal(y)) \vee (Phone(x) \wedge Literal(y))$
$\forall x \bullet \exists y \bullet inCity(x,y) \Rightarrow Address(x) \wedge City(y)$
$\forall x \bullet \exists y \bullet hasPhone(x,y) \Rightarrow Address(x) \wedge Phone(y)$
[From edge(u_4, u_7)***]***
$\forall x, y, z \bullet inCity(x,y) \wedge inCity(x,z) \wedge Address(x) \wedge City(y) \wedge City(z) \Rightarrow y = z\}$

7.5. Construcción de las descripciones semánticas

Al principio de la sección anterior, presentamos informalmente la relación existente entre una descripción semántica y un individual tree. Ahora la formalizaremos, las descripciones semánticas se construyen a partir de individual trees, colapsandolos con objeto de inferir la información acerca de la semántica detrás de los individuos. Dos conceptos nos permiten dicha formalización: vértices colapsables y rutas colapsables.

7.5.1. Vértices colapsables

Definición 7.3 (vértices colapsables) *Un conjunto de vértices en un árbol etiquetado es colapsable, si todos los vértices en el conjunto tienen un mismo padre, los ejes*

que relacionan el padre común con dichos vértices están etiquetados con un mismo nombre de propiedad y no existe otro vértice que pertenezca al conjunto que cumpla las restricciones anteriores.

El predicado *collapsableVertices* formaliza la definición anterior. El predicado se satisface si un conjunto de vértices en un árbol etiquetado es colapsable.

$$
\begin{array}{|l}
collapsableVertices : LabelledTree \times \mathbb{P}\, Vertex \\
\hline
\forall\, lt : LabelledTree;\ sv : \mathbb{P}\, Vertex \bullet collapsableVertices(lt, sv) \Leftrightarrow \\
\quad \exists\, v : Vertex;\ pn : PropertyName \mid sv \subseteq children(lt.t, v) \bullet \\
\quad\quad \forall\, v_x : sv \bullet lt.edgeType(v, v_x) = pn \bigwedge \\
\quad\quad \forall\, v_x : children(lt.t, v) \setminus sv \bullet lt.edgeType(v, v_x) \neq pn
\end{array}
$$

El árbol etiquetado de un individual tree puede tener vértices colapsables, sin embargo el de una descripción semántica no los tiene. Por ejemplo, los conjunto de vértices $\{v_4, v_5\}$ y $\{v_{12}, v_{13}\}$ de la Figura §7.2 son colapsables; esto significa que las propiedades *hasAddress* y *hasPhone* tienen cardinalidad múltiple. La misma información se da en la descripción semántica de la Figura §7.3, al indicar que los vértices u_4 y u_8 están en el conjunto *repeated* de la descripción semántica.

7.5.2. Rutas colapsables

Definimos una ruta en un árbol etiquetado como una secuencia de vértices, de manera que todo vértice está conectado en el árbol al siguiente vértice en la secuencia. Solo consideraremos las rutas que conectan la raíz del árbol con las hojas. Cada ruta tiene asociada un patrón, que es una secuencia alternada de tipo de vértice y tipo de eje, de manera que, los vértices en una ruta son sustituidos por su tipo, y entre cada dos vértices añadimos el tipo del eje que los conecta en el árbol. Por ejemplo, la ruta $\langle v_1, v_4, v_5 \rangle$ del *labelledTree* en la Figura §7.2, tiene asociada el patrón $\langle Restaurant, hasAddress, Address, street, FILLER \rangle$. Formalmente, las rutas y patrones se definen como tipos de datos. Además, la función *pattern* toma como entrada un árbol etiquetado y una ruta en ese árbol y devuelve el patrón asociado a la ruta.

$$Path == \text{seq}\, Vertex$$
$$PathPattern == \text{seq}(ConceptName \cup PropertyName)$$

$$
\begin{array}{|l}
pattern : LabelledTree \times Path \nrightarrow PathPattern \\
\hline
\forall\, lt : LabelledTree;\ p : Path;\ q : PathPattern \mid p \in paths(lt.t) \bullet pattern(lt, p) = q \Leftrightarrow \\
\quad \#q = 2 * \#p - 1 \land \\
\quad \forall\, i : 1 \,..\, \#p \bullet q(2 * i - 1) = lt.vertexType(p(i)) \land \\
\quad \forall\, i : 1 \,..\, \#p - 1 \bullet q(2 * i) = lt.edgeType(p(i), p(i+1)))
\end{array}
$$

Diferentes rutas pueden encajar en un mismo patrón. Definimos una relación binaria de igualdad que relaciona rutas basándose en el patrón que siguen dentro de un árbol etiquetado. La relación es especificada de la siguiente manera:

$$
\begin{array}{|l}
\_ \sim \_ : (LabelledTree \times Path) \leftrightarrow (LabelledTree \times Path) \\
\hline
\forall\, lt_1, lt_2 : LabelledTree;\ p_1, p_2 : Path;\ (lt_1, p_1) \sim (lt_2, p_2) \Leftrightarrow \\
\quad pattern(lt_1, p_1) = pattern(lt_2, p_2)
\end{array}
$$

Lema 7.1 *La relación binaria de igualdad es uan relación de equivalencia en el conjunto Path.*

Demostración Para probar el lema necesitamos demostrar que $\sim$ es reflexiva, simétrica y transitiva.

Reflexividad. Para cualquier $(lt, p) : LabelledTree \times Path$; se cumple que $pattern(lt, p) = pattern(lt, p)$; por lo tanto $(lt, p) \sim (lt, p)$.

Simetría. Sea $(lt_1, p_1), (lt_2, p_2) : LabelledTree \times Path$; se cumple que

$$
\begin{array}{ll}
 & (lt_1, p_1) \sim (lt_2, p_2) \\
\Leftrightarrow & \qquad \text{[por definición de } \sim\text{]} \\
 & pattern(lt_1, p_1) = pattern(lt_2, p_2) \\
\Leftrightarrow & \qquad \text{[simetría de la igualdad en el lenguaje de los conjuntos]} \\
 & pattern(lt_2, p_2) = pattern(lt_1, p_1) \\
\Leftrightarrow & \qquad \text{[por definición de } \sim\text{]} \\
 & (lt_2, p_2) \sim (lt_1, p_1)
\end{array}
$$

Transitividad. Sea $(lt_1, p_1), (lt_2, p_2), (lt_3, p_3) : LabelledTree \times Path$; se cumple que

$$
\begin{array}{ll}
 & (lt_1, p_1) \sim (lt_2, p_2) \land (lt_2, p_2) \sim (lt_3, p_3) \\
\Leftrightarrow & \qquad \text{[por definición de } \sim\text{]} \\
 & pattern(lt_1, p_1) = pattern(lt_2, p_2) \land pattern(lt_2, p_2) = pattern(lt_3, p_3) \\
\Leftrightarrow & \qquad \text{[transitividad de la igualdad en el lenguaje de los conjuntos]} \\
 & pattern(lt_1, p_1) = pattern(lt_3, p_3) \\
\Leftrightarrow & \qquad \text{[por definición de } \sim\text{]} \\
 & (lt_1, p_1) \sim (lt_3, p_3)
\end{array}
$$

□

Definición 7.4 (Rutas colapsables) *Un conjunto de rutas es colapsable si todas sus rutas están relacionadas por la relación* $\sim$.

La relación de igualdad nos permite definir una partición de un conjunto de rutas en clases equivalentes (todos los elementos que son equivalentes entre sí están en una misma clase). Cada clase se corresponde con un conjunto de rutas colapsables. Por tanto, las rutas colapsables en un árbol etiquetado vienen definidas por el conjunto cociente que se obtiene a partir de la relación $\sim$.

$$
\begin{array}{|l}
\_/\sim : \mathit{LabelledTree} \rightarrow \mathbb{P}\,\mathbb{P}\,\mathit{Path} \\
\hline
\forall\, lt : \mathit{LabelledTree};\ ssp : \mathbb{P}\,\mathbb{P}\,\mathit{Path};\ sp : \mathbb{P}\,\mathit{Path} \mid sp = \mathit{paths}(lt) \bullet lt/\sim = ssp \Leftrightarrow \\
\quad (sp = \bigcup ssp) \land \\
\quad (\forall\, p : sp \bullet \nexists pp, pq : ssp \mid pp \neq pq \bullet p \in pp \land p \in pq) \land \\
\quad (\forall\, pp : ssp \bullet (\forall\, p, q : pp \bullet (lt, p) \sim (lt, q)) \land (\nexists pq : ssp, q : pq \mid pp \neq pq \bullet (lt, p) \sim (lt, q))
\end{array}
$$

Ejemplo 7.4 *La Figura §7.5 muestra la partición de un árbol etiquetado en rutas colapsables. El árbol etiquetado se corresponde al del individual tree de la figura §7.2.*

El árbol etiquetado de un individual tree puede tener rustas colapsables, sin embargo, el árbol etiquetado de una descripción semántica no tiene. Por ejemplo, las rutas $\langle v_1, v_4, v_6\rangle$ y $\langle v_1, v_5, v_9\rangle$ en la Figura §7.2 son colapsables, ya que siguen el mismo patrón: $\langle \mathit{Restaurant}, \mathit{hasAddress}, \mathit{Address}, \mathit{number}, \mathit{FILLER}\rangle$. En la descripción semántica de la Figura §7.3, etán colapsadas en la ruta $\langle u_1, u_4, u_5\rangle$.

Lema 7.2 *La partición de un árbol etiquetado en rutas colapsables es única.*

Demostración La prueba es directa ya que las rutas colapsables son equivalentes y se han definido por el conjunto cociente de una relación de equivalencia [11]. □

Teorema 7.1 *Si un árbol etquetado no tiene vértices colapsables, entonces tampoco tiene rutas colapsables.*

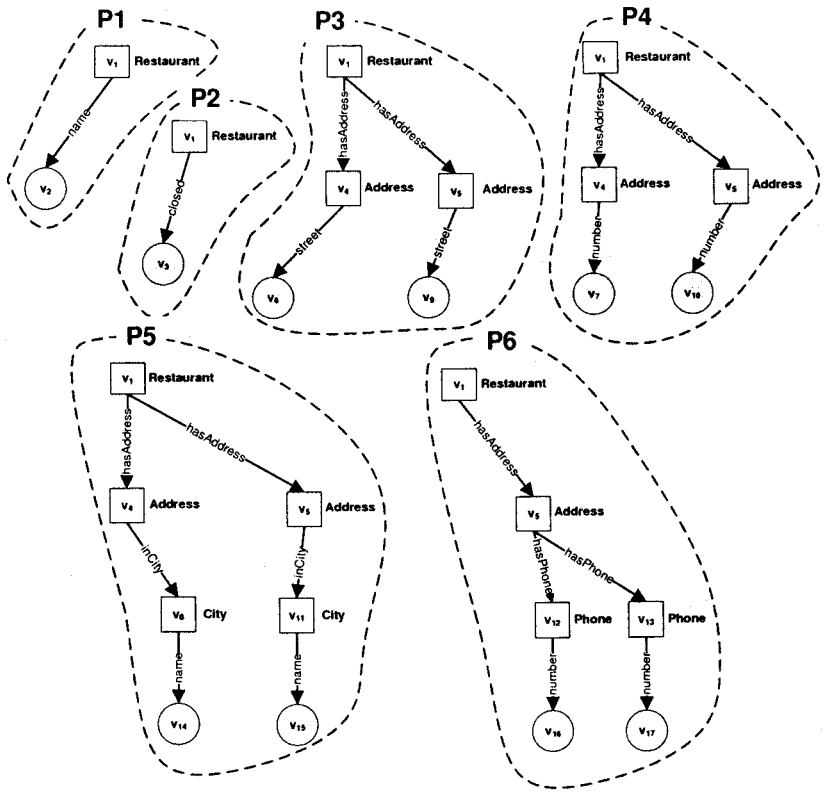


Figura 7.5: *Partición de un árbol etiquetado en rutas colapsables.*

Demostración Supongamos que existe it : *IndividualTree* con rutas colapsables, entonces existirá al menos dos rutas distintas p_1 y p_2 en $it.lt$, de manera que $(lt, p_1) \sim (lt, p_2)$. De acuerdo con la definición de vértices colapsables, no puede existir un vértice v en p_1 y p_2 de manera que $p_1(i) = v \wedge p_2(i) = v \wedge p_1(i+1) \neq p_2(i+1)$, por lo tanto $p_1 = p_2$. Lo que contradice nuestra suposición inicial de que it no tiene vértices colapsables y si tiene rutas colapsables. De esta manera concluimos que la proposición original debe ser cierta. □

Observe que la proposición necesaria no es siempre verdad. Por ejemplo, si eliminamos los vértices v_7, v_8, v_{14} y v_9 del individual tree de la Figura §7.2, no tendríamos rutas colapsables, pero si vértices colapsables (v_4 y v_5).

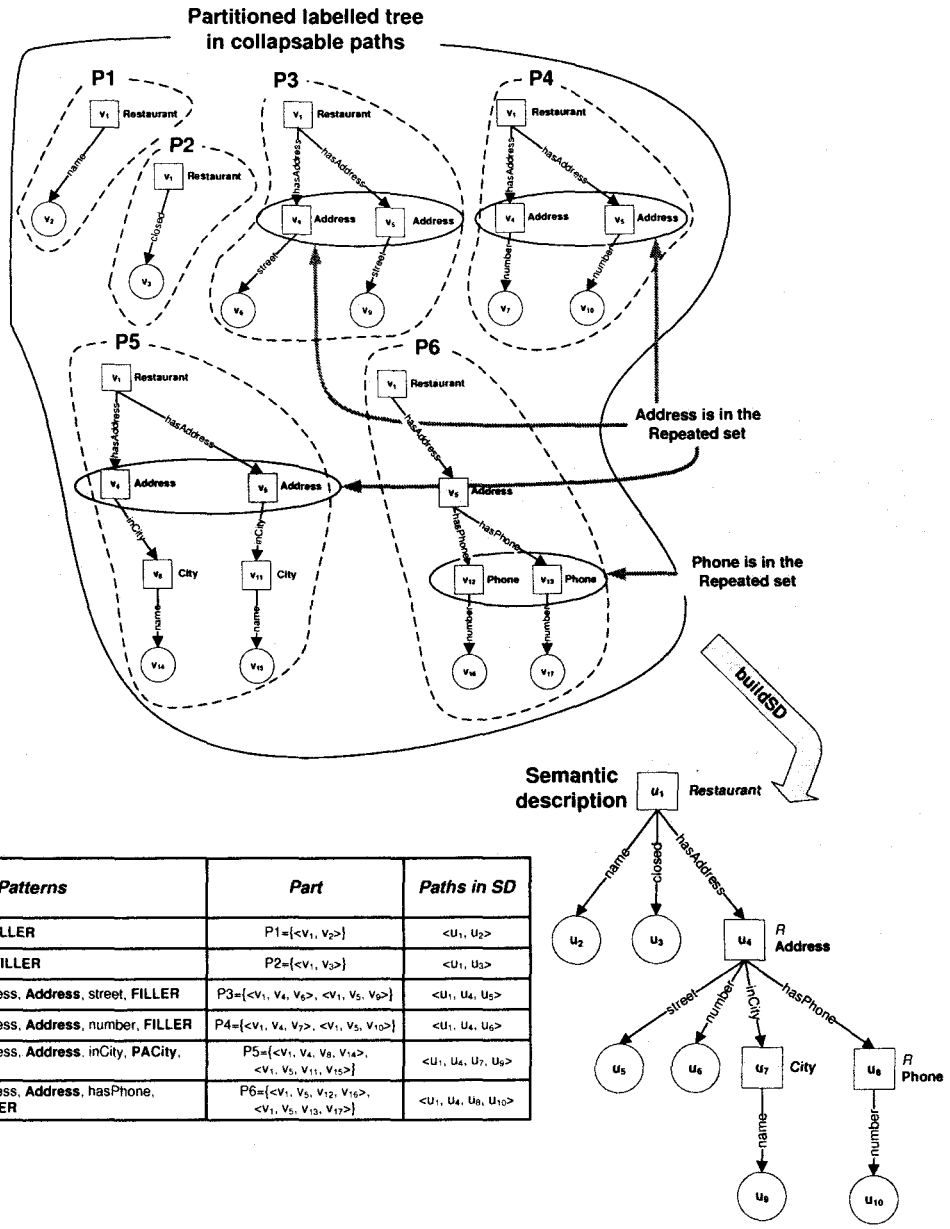
7.5.3. Colapsamiento de individual trees

La Función *buildSD* toma a la entrada un individual tree y devuelve una descripción semántica. Para ello, colapsa los vértices y rutas del árbol etiquetado del individual tree. El colapsamiento se lleva a cabo definiendo una función biyectiva total que relaciona conjunto de rutas colapsables a una única ruta en la descripción semántica. Los vértices colapsables son detectados y añadidos al conjunto *repeated* de la descripción semántica.

$$
\begin{array}{|l}
buildSD : IndividualTree \rightarrow SemanticDescription \\
\hline
\forall\, it : IndividualTree;\ sd : SemanticDescription \bullet buildSD(it) = sd \Leftrightarrow \\
\quad \exists f : it.lt/\!\sim \twoheadrightarrow paths(sd.lt.t) \bullet \\
\quad\quad \forall\, pp : \mathrm{dom} f;\ p : pp \bullet (it.lt, p) \sim (sd.lt, f(pp)) \wedge \\
\quad\quad \forall\, pp : \mathrm{dom} f;\ p, q : pp;\ n : 2\,..\,\#p \mid p(i) \neq q(i) \wedge p(i-1) = q(i-1) \bullet \\
\quad\quad\quad f(pp)(i) \in sd.repeated
\end{array}
$$

Ejemplo 7.5 *La Figura §7.6 ilustra como funciona buildSD. Muestra las rutas colapsables, a la ruta a la que se asocian en la descripción semántica y los vértices en el conjunto repeated.*

La Función *buildSD* nos permite construir descripciones semánticas a partir de un único individual tree. Sin embargo, un único individual tree no tiene porque hacer uso de toda la información semántica necesaria para dar significado a todos los individuos ofrecidos por el sitio web. Denominamos descripción semántica global a aquella que contiene la información semántica de todos los individuos que ofrece un sitio web. La Función *mergeSDs* nos permite obtener una descripción semántica más completa a partir de un conjunto de descripciones semánticas.



Patterns	*Part*	*Paths in SD*
Restaurant, name, **FILLER**	P1={<v_1, v_2>}	<u_1, u_2>
Restaurant, closed, **FILLER**	P2={<v_1, v_3>}	<u_1, u_3>
Restaurant, hasAddress, **Address**, street, **FILLER**	P3={<v_1, v_4, v_6>, <v_1, v_5, v_9>}	<u_1, u_4, u_5>
Restaurant, hasAddress, **Address**, number, **FILLER**	P4={<v_1, v_4, v_7>, <v_1, v_5, v_{10}>}	<u_1, u_4, u_6>
Restaurant, hasAddress, **Address**, inCity, **PACity**, name, **FILLER**	P5={<v_1, v_4, v_8, v_{14}>, <v_1, v_5, v_{11}, v_{15}>}	<u_1, u_4, u_7, u_9>
Restaurant, hasAddress, **Address**, hasPhone, **Phone**, number, **FILLER**	P6={<v_1, v_5, v_{12}, v_{16}>, <v_1, v_5, v_{13}, v_{17}>}	<u_1, u_4, u_8, u_{10}>

Figura 7.6: *La función builSD.*

$$
\begin{array}{|l}
mergeSDs : \mathbb{P}\, SemanticDescription \rightarrow SemanticDescription \\
\hline
\forall\, ssd : \mathbb{P}\, SemanticDescription;\ sd : SemanticDescription \bullet mergeSDs(ssd) = sd \Leftrightarrow \\
\quad \forall\, sd_x : ssd;\ p : paths(sd_x.lt.t) \bullet \exists_1\, q : paths(sd.lt.t) \bullet \\
\qquad (sd_x.lt, p) \sim (sd.lt, q) \bigwedge (\forall\, i : 1 \,..\, \#p \mid p(i) \in sd_x.repeated \bullet q(i) \in sd.repeated)) \bigwedge \\
\quad \forall\, p : paths(sd.lt.t) \bullet \exists\, sd_x : ssd;\ q : paths(sd.lt.t) \bullet (sd_x.lt, p) \sim (sd.lt, q) \bigwedge \\
\quad \forall\, p : paths(sd.lt.t);\ i : 1 \,..\, \#p \mid p(i) \in sd.repeated \bullet \\
\qquad \exists\, sd_x : ssd;\ q : Path \mid q \in paths(sd.lt.t) \bullet q(i) \in sd_x.repeated
\end{array}
$$

Sea $\mathcal{I}$ el conjunto de todos los individuos en un sitio web y $\mathcal{SD}$ una descripción semántica global, entonces $\mathcal{SD} = buidGlobalSD(\mathcal{I})$; donde la Función *buidGlobalSD* se define como el resultado de mezclar todas las descripciones semánticas obtenidas para los elementos en $\mathcal{I}$:

$$
\begin{array}{|l}
buidGlobalSD : \mathbb{P}\, IndividualTree \rightarrow SemanticDescription \\
\hline
\forall\, pit : \mathbb{P}\, IndividualTree \bullet MergeSDs(\{it : pit \bullet buildSD(sd)\})
\end{array}
$$

7.6. Relación de la información con las descripciones semánticas

Según las suposiciones de la Sección §7.2, cada vértice de tipo FILLER en una descripción semántica representa a un atributo en el *HierarchicalSlot* que aparece en un determinado nivel jerárquico. Por tanto, la información extraída puede ser relacionada unívocamente con una descripción semántica proporcionando la información de localización para los vértices de tipo *FILLER*. El atributo que un vértice de tipo *FILLER* representa puede ser referenciado por un nivel jerárquico, y por un índice natural que lo distingue de todos los vértices en un mismo nivel jerárquico. Por ejemplo el vértice u_2 en la descripción semántica de la Figura §7.7 representa el primer atributo de los vértices en el nivel jerárquico $H0$; y u_8 representa el tercer atributo de los vértices en el nivel jerárquico $H1$. Dos conceptos nos permiten formalizar y presentar la manera de obtener la información de localización: áreas de influencia y áreas de influencia reflejadas.

7.6.1. Áreas de influencias y áreas de influencia reflejadas

Cada vértice de tipo concepto en el conjunto *repeated* de una descripción semántica o vértice raíz, define un área de influencia, de manera que, todos

los vértices de tipo *FILLER* en un mismo área de influencia toman valores de un mismo nivel jerárquico.

Definición 7.5 (Áreas de influencia) *El área de influencia de un vértice v que está en el conjunto repeated o es la raíz del árbol etiquetado de una descripción semántica, está compuesto por todos aquellos vértices que son alcanzables desde v sin pasar por otro vértice del conjunto repeated. Formalmente:*

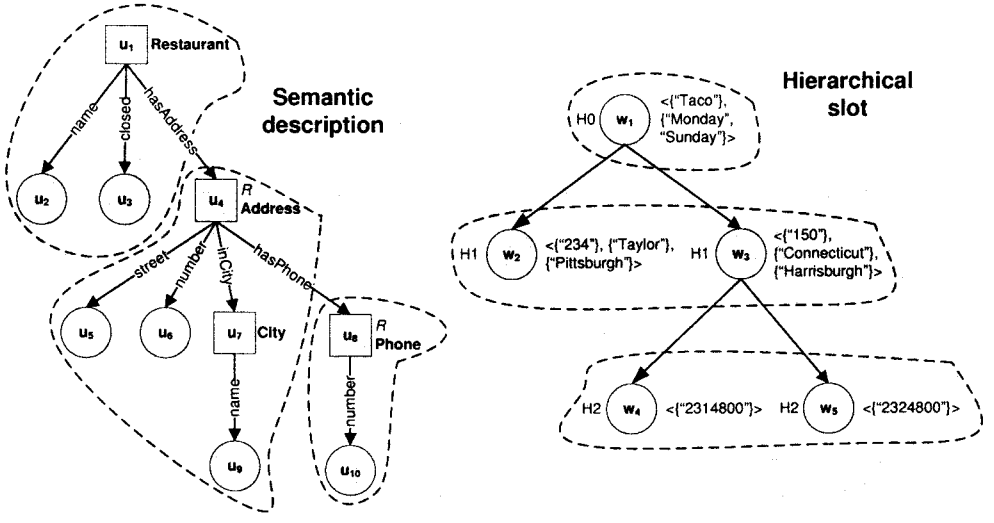
$$
\begin{array}{|l}
influenceArea : SemanticDescription \times \mathbb{P}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ sv : \mathbb{P}\, Vertex \bullet influenceArea(sd, sv) \Leftrightarrow \\
\quad \exists_1\, v : sv \mid v \in sd.repeated \lor v = root(sd.lt.t) \bullet sv \setminus \{v\} = \\
\quad\quad \{p : paths(sd.lt.t);\ n, i : \mathbb{N} \mid p(n) = v \land i \in n + 1 \,..\, \#p \land \\
\quad\quad\quad (\nexists j : n + 1 \,..\, i \bullet p(j) \in sd.repeated) \bullet p(i)\}
\end{array}
$$

Ejemplo 7.6 *La Figura §7.7 muestra una partición de la descripción semántica en áreas de influencia. Hay tres áreas diferentes, definidas por la raíz (Restaurant) y los vértices en el conjunto repeated (Address y Phone). El área definido por Restaurant está asociado al nivel jerárquico H0; los vértices de tipo FILLER en ese área toman su valor de los vértices que se encuentran en el nivel H0 del HierarchicalSlot*

Dado que las descripciones semánticas se obtienen de los individual trees de tal manera que los vértices y ejes en el individual tree son reflejados a vértices y ejes en las descripciones semánticas, las áreas de influencia de una descripción semántica se reflejan en los individual trees.

Definición 7.6 (Áreas de influencia reflejadas) *El área de influencia reflejada de un vértice v del árbol etiquetado de un individual tree, que está asociado a un vértice perteneciente al conjunto repeated de la descripción semántica o es la raíz, está compuesto de todos los vértices alcanzables desde v sin pasar por ningún otro vértice que esté reflejado en un vértice del conjunto repeated de la descripción semántica.*

$$
\begin{array}{|l}
mirroredInfluenceArea : SemanticDescription \times IndividualTree \times \mathbb{P}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ it : IndividualTree;\ sv : \mathbb{P}\, Vertex \bullet \\
\quad mirroredInfluenceArea(sd, it, sv) \Leftrightarrow \\
\quad\quad \exists_1\, su : sd.lt.t.vertices \mid influenceArea(sd, su) \bullet \\
\quad\quad\quad (\forall\, v : sv \bullet mirrorInSD(sd, it, v) \in su \land \\
\quad\quad\quad \nexists v \in it.lt.t.vertices \mid v \notin sv \bullet mirrorInSD(sd, it, v) \in su)
\end{array}
$$



Influence areas	Level on hierarchical slot
$\{u_1, u_2, u_3\}$	H0
$\{u_4, u_5, u_6, u_7, u_9\}$	H1
$\{u_8, u_{10}\}$	H2

Figura 7.7: *Áreas de influencia.*

La Función *mirrorInSD* toma a la entrada una descripción semántica, un individual tree y un vértice (perteneciente al árbol etiquetado del individual tree) y devuelve el vértice reflejado en la descripción semántica.

$$
\begin{array}{|l}
mirrorInSD : SemanticDescription \times IndividualTree \times Vertex \nrightarrow Vertex \\
\hline
\forall\, sd : SemanticDescription;\ it : IndividualTree;\ v, u : Vertex \bullet \\
\quad mirrorInSD(sd, it, v) = u \Leftrightarrow \\
\qquad \exists p_1 : paths(lt.lt.t);\ p_2 : paths(sd.lt.t);\ i : \mathbb{N} \mid p_1(i) = v \bigwedge (sd.lt, p_1) \sim (it.lt, p_2) \bullet \\
\qquad\quad p_2(i) = u
\end{array}
$$

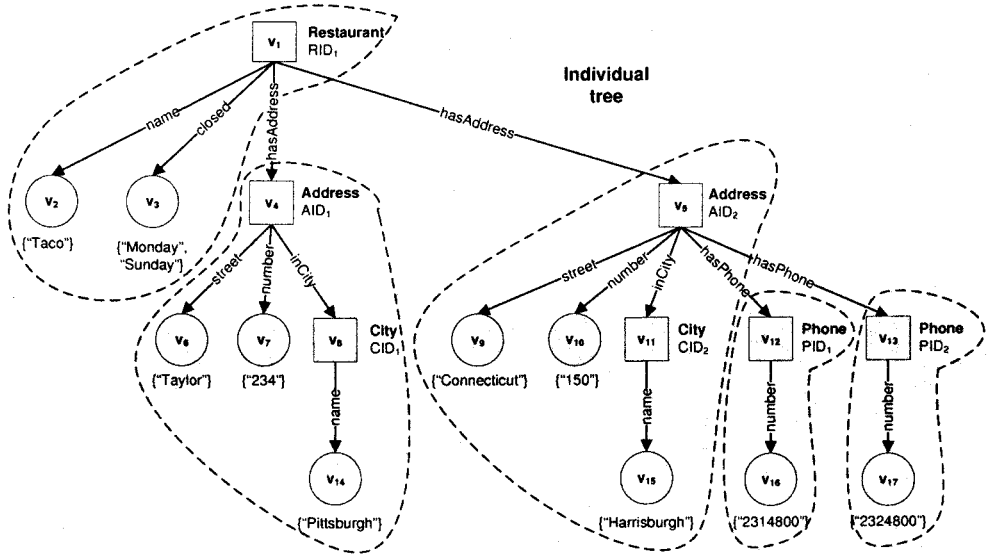
Ejemplo 7.7 *La Figura §7.8 ilustra una partición del individual tree en áreas de influencia reflejadas. Existen cinco áreas definidas por la raíz (Restaurant), y los vértices que tienen como reflejado un vértice en el conjunto repeated de la descripción semántica: (Address and Phone). El área definida por Restaurant tiene asociada el nivel jerárquico H0 y los vértices de tipo FILLER de éste área toman valores del vértice w_0 del HierarchicalSlot. Las áreas definidas por v_4 y v_5, representan instancias del concepto dirección y están asociadas a los vértices w_2 y w_3 del HierarchicalSlot; dado que el concepto Address en la descripción semántica está asociado al nivel jerárquico H1, w_2 y w_3 están etiquetados con H_1. Omitimos el resto de los detalles ya que no debería ser problemático para el lector.*

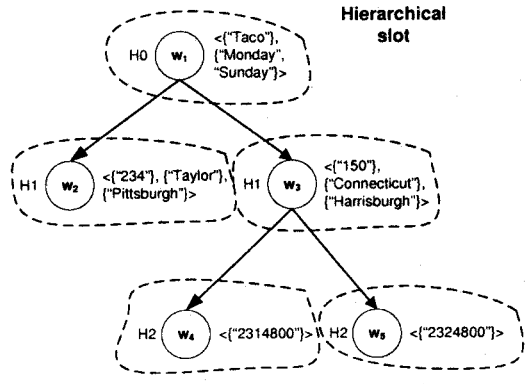
7.6.2. Construcción de *Location*

De acuerdo a las ideas anteriores, definimos el tipo de datos *Location* que asocia a los vértices de una descripción semántica de tipo *FILLER*, niveles jerárquicos (*vertexLevel*) e índices naturales (*vertexPosition*) que unívocamente indican el atributo dento de un mismo nivel jerárquico.

$$
\begin{array}{|l}
Location \\
\hline
vertexPosition : Vertex \nrightarrow \mathbb{N} \\
vertexLevel : Vertex \nrightarrow Label \\
\hline
\end{array}
$$

Antes de especificar como se obtiene la información de localización, formalizaremos la relación existente entre un individual tree, un *HierarchicalSlot*, una descripción semántica global y la función de localización. El predicado *translation* relaciona los elementos anteriores. Está basado en la idea de que una descripción semántica puede ser vista como un contenedor de patrones semánticos, donde cada patrón viene definido por un área de influencia; los *HierarchicalSlot* materializan las áreas de influencia reflejadas de esos patrones. De esta manera, *translation* usa una función biyectiva total que asocia vértices





Reflected influence areas	Vertex on hierarchical slot
$\{v_1, v_2, v_3\}$	w_1
$\{v_4, v_6, v_7, v_8, v_{14}\}$	w_2
$\{v_5, v_9, v_{10}, v_{11}, v_{15}\}$	w_3
$\{v_{12}, v_{16}\}$	w_4
$\{v_{13}, v_{17}\}$	w_5

Figura 7.8: *Áreas de influencia reflejadas.*

del *HierarchicalSlot* con áreas de influencias reflejadas en el individual tree, de forma que todos los vértices de tipo *FILLER* en el individual tree, toman sus atributos de la información de localización asociada a su vértice reflejado en la descripción semántica. También, a los ejes en el *HierarchicalSlot* se le asocian ejes del individual tree que conectan vértices que definen áreas de influencia reflejadas (los tipos de estos ejes se obtienen de la descripción semántica). La función auxiliar *subtreeRoot* está definida en el Apéndice §A; ésta función devuelve la raíz de un subárbol representado por un conjunto de vértices.

$$
\begin{array}{l}
translation : IndividualTree \times HierarchicalSlot \times SemanticDescription \times Location \\
\hline
\forall\, it : IndividualTree;\ hs : HierarchicalSlot;\ sd : SemanticDescription;\ loc : Location \bullet \\
\quad translation(it, hs, sd, loc) \Leftrightarrow \exists f : hs.t.vertices \twoheadrightarrow \mathbb{P}\, it.lt.t.vertices \bullet \\
\qquad \forall sv : \mathrm{ran} f \bullet mirroredInfluenceAreas(sd, it, sv) \land \\
\qquad \forall w : \mathrm{dom} f;\ v : Vertex \mid v \in f(w) \land it.lt.vertexType(v) = FILLER \bullet \\
\qquad\quad it.vertexAttribute(v) = \\
\qquad\qquad hs.vertexAttributes(w)(loc.vertexPosition(mirrorInSD(sd, it, v))) \land \\
\qquad\quad loc.vertexLevel(mirrorInSD(sd, it, v)) = hs.vertexHLevel(w))) \land \\
\qquad \forall (w_1, w_2) : hs.t.edges \bullet \exists_1 v_1, v_2 : Vertex \bullet \\
\qquad\quad v_1 = subtreeRoot(it.lt.t, f(w_1)) \land v_2 = subtreeRoot(it.lt.t, f(w_2)) \land \\
\qquad\quad (v_1, v_2) \in it.lt.edges \land \\
\qquad\quad sd.lt.edgeType(mirrorInSD(sd, it, v_1), mirrorInSD(sd, it, v_2)) = \\
\qquad\qquad it.lt.edgeType(v_1, v_2)
\end{array}
$$

Ejemplo 7.8 *La figura §7.9 ilustra gráficamente dos ejemplos de las relaciones existentes entre IndividualTree, HierarchicalSlot, la descripción semántica global y la información de localización. Una X en la información de localización representa cualquier valor (no es de interés en el ejemplo).*

Sea $\mathcal{IH}$ todas las parejas de *IndividualTree* y *HierarchicalSlot* que nos ofrece un sitio web y $\mathcal{SD}$ la descripción semántica global, entonces la información de localización $\mathcal{LOC}$ se obtiene como: $\mathcal{LOC} = buildLocation(\mathcal{IH}, \mathcal{SD})$, donde la función *buildLocation* se define a partir del predicado *translation*, asegurando que la información de localización está bien definida para cada elemento en $\mathcal{IH}$:

$$
\begin{array}{l}
buildLocation : \mathbb{P}(IndividualTree \times HierarchicalSlot) \times SemanticDescription \rightarrow Location \\
\hline
\forall\, ih : \mathbb{P}(IndividualTree \times HierarchicalSlot);\ sd : SemanticDescription;\ loc : Location \bullet \\
\quad buildLocation(ih, sd) = loc \Leftrightarrow \\
\qquad \mathrm{dom}\, loc.vertexPosition = \{u : sd.lt.t.vertices \mid sd.lt.vertexType(u) = FILLER \bullet u\} \land \\
\qquad \mathrm{dom}\, loc.vertexLevel = \{u : sd.lt.t.vertices \mid sd.lt.vertexType(u) = FILLER \bullet u\} \land \\
\qquad \forall (it, hs) : ih \bullet translation(it, hs, sd, loc)
\end{array}
$$

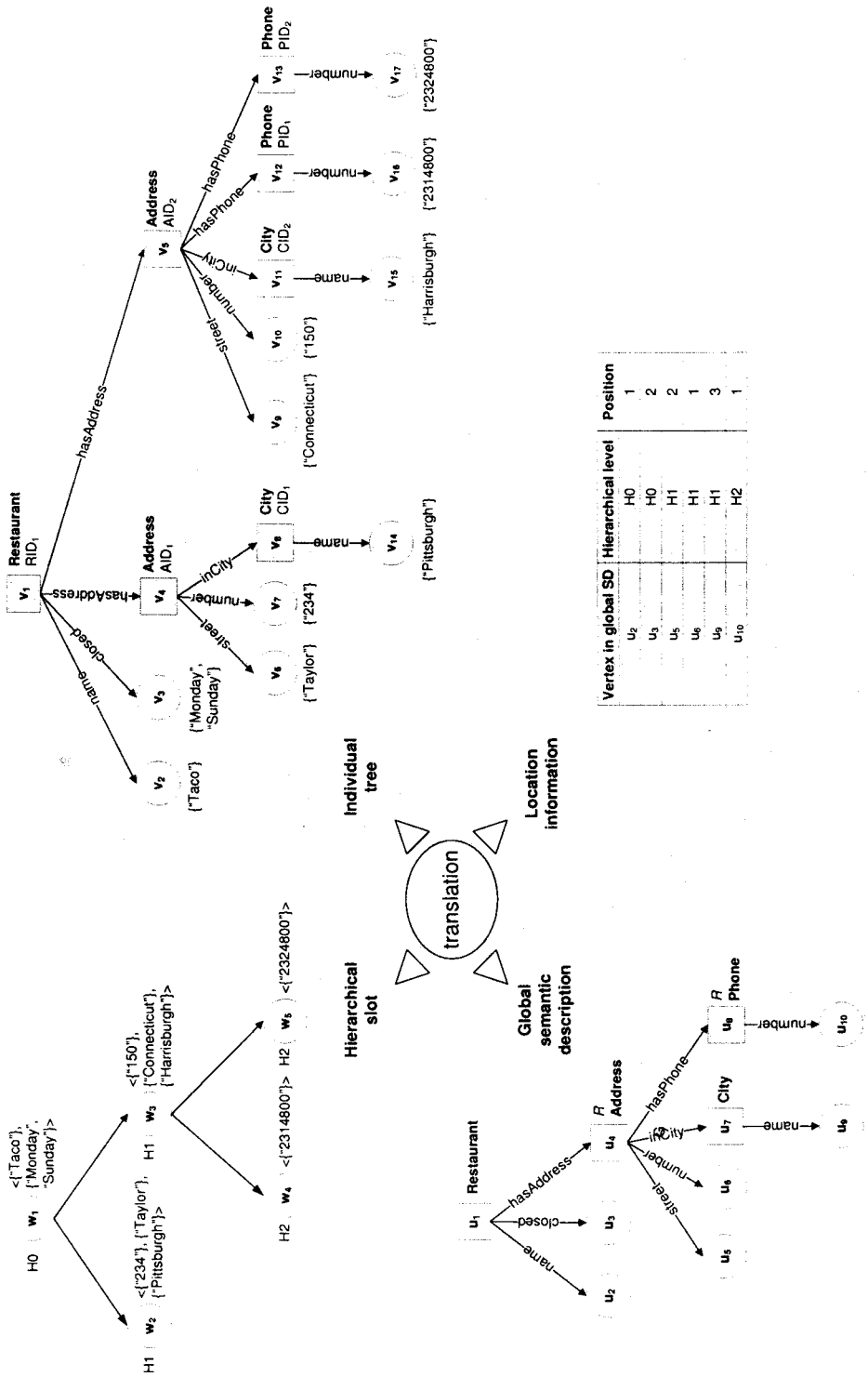
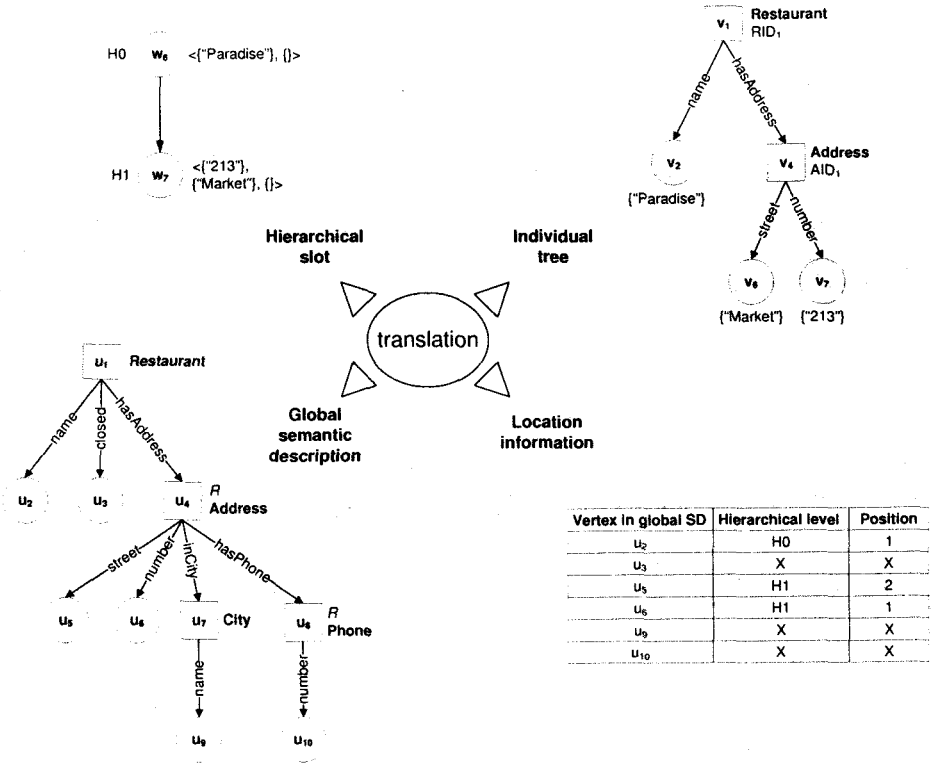


Figura 7.9: *Relación de traducción.*



Vertex in global SD	Hierarchical level	Position
u_2	H0	1
u_3	X	X
u_5	H1	2
u_6	H1	1
u_9	X	X
u_{10}	X	X

Figura 7.9: *Relación de traducción (bis).*

7.7. Traductores semánticos

El problema de la traducción semántica se soluciona definiendo un algoritmo genérico, llamado traductor semántico, que utiliza la descripción semántica global y la información de localización obtenidas previamente para dar automáticamente semántica a cualquier *StructuredInformation* extraída por el wrapper. Debido a que *StructuredInformation* es un conjunto de *HierarchicalSlot*, el traductor semántico traduce cada árbol en el conjunto; además, utiliza *toAbox* para convertir el *IndividualTree* obtenido a un *Abox* (la función *toAbox* está definida en el apéndice §B).

$$semanticTranslator : SemanticDescription \times Location \times StructuredInformation \rightarrow Abox$$

$$\forall\, sd : SemanticDescription;\ loc : Location;\ si : StructuredInformation;\ a : Abox \bullet$$
$$semanticTranslator(sd, loc, si) =$$
$$\bigcup\{\forall\, hs : si;\ it : IndividualTree \mid translation(it, hs, sd, loc) \bullet toAbox(it)\}$$

Capítulo 8

Materialización del problema de traducción semántica

The worst thing one can do is not to try, to be aware of what one wants and not give in to it, to spend years in silent hurt wondering if something could have materialised - never knowing.

Dr. David Viscott, 1938–
American psychiatrist

En este capitulo aplicamos los resultados del capítulo anterior para ofrecer una posible implementación de tres algoritmos que resuelven el problema de traducción semántica. El Capítulo está organizado de la siguiente manera: la Sección §8.1 es la introducción; la Sección §8.2 presenta un algoritmo para la construcción de descripciones semánticas; la Sección §8.3 presenta un algoritmo para la obtención de la información de localización; finalmente, la Sección §8.4 presenta un algoritmo genérico e independiente del dominio para la traducción semántica.

8.1. Introducción

El nivel de abstracción con el que se ha presentado el marco de trabajo nos proporciona un máximo grado de libertad de implementación. En éste capítulo presentamos una posible implementación para el problema de traducción semántica. La implementación consiste en tres algoritmos: el primero obtiene la descripción semántica global; el segundo nos permite obtener la información de localización; el tercero es un algoritmo genérico independiente del dominio que implementa un traductor semántico. También probamos que esos algoritmos son implementaciones correctas con respecto a la especificación presentada en el Capítulo §7.

La implementación viene descrita con reglas de transición de Plotkin, dado que se ha probado que esta técnica es a la vez simple y potente [73]. Nuestro principal objetivo no es dar una implementación óptima, sino demostrar que el marco de trabajo puede ser implementado usando máquinas de estado finitas. No obstante, señalaremos algunas ideas que ayudan a optimizar la implementación, pero no daremos muchos detalles dado que éstas caen fuera del alcance de esta memoria.

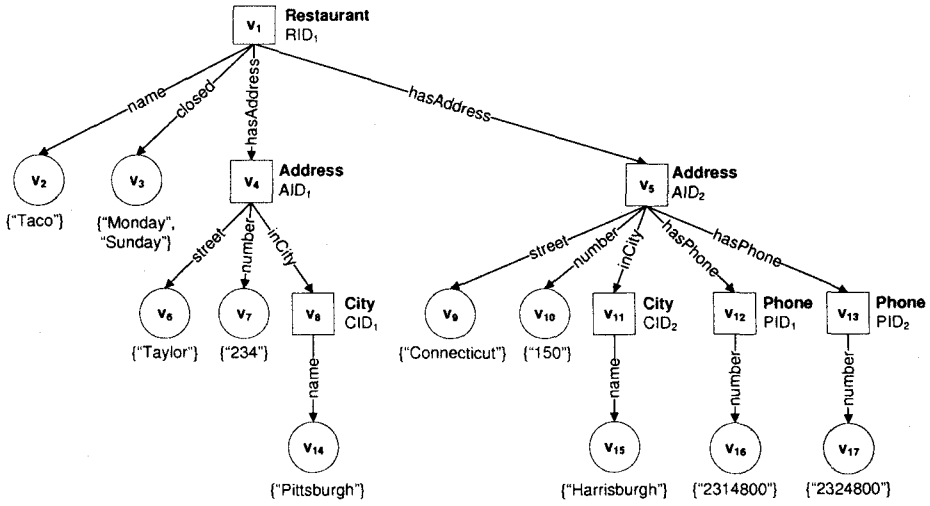
8.2. Construcción de una descripción semántica

En el Capítulo anterior, explicamos cómo una descripción semántica global se construye uniendo la información semántica que nos proporciona todos los individuos en el sitio web. Los métodos inductivos parecen ser apropiados para implementar un algoritmo que nos resuelva este problema. Sin embargo, en esta implementación, seguimos una estrategia más sencilla, que requiere que los usuarios tengan una buena comprensión de las relaciones semánticas que aparecen en el sitio web. Esto implica que el usuario debe de construir un Abox con información acerca de sólo un individuo que cumpla los siguientes requisitos:

Requisito 1. Todos los nombres de propiedades y conceptos necesarios para expresar la semántica de la información en el sitio web, aparecen en el Abox.

Requisito 2. Cada propiedad de cardinalidad múltiple ($[0\,..\,n]$ o $[1\,..\,n]$) debe de aparecer al menos dos veces.

Requisito 3. Cada propiedad opcional ($[0\,..\,1]$) aparece una vez.



(a) Individual tree (repetición de la figura §7.2)

Name: Taco **Close on:** Monday & Sunday
Address: 234 Taylor
Pittsburgh

Address: 150 Connecticut
Harrisburgh
Phone: 2314800
Phone: 2324800

Name: Paradise
Address: 213 Market

Name: FireMeal **Close on:** Monday
Address: 122 West
New York
Phone: 2344800

(b) Página web (repetición de la figura §6.2(a))

Figura 8.1: *Requisitos del algoritmo* `buildSD`.

Los restricciones anteriores aseguran que el Abox contenga toda la información semántica necesaria para la traducción de todos los individuos del sitio web. Por conveniencia, el algoritmo se presenta en términos de individual trees en vez de Aboxes (como se mencionó en el capítulo anterior, son notaciones equivalentes para representar individuos).

Ejemplo 8.1 *El individual tree de la Figura §8.1(a) no cumple los requisitos anteriores, ya que la dirección* AID_1 *no tiene números de teléfonos (debería tener más de uno ya que la propiedad hasPhone tiene cardinalidad múltiple de acuerdo a la información que provee la página web de la Figura §8.1(b)).*

8.2.1. Algoritmo

El algoritmo `buildSD` toma a la entrada un individual tree que cumple las restricciones anteriores y devuelve una descripción semántica. Funciona de la siguiente forma: inicialmente, el árbol etiquetado de la descripción semántica es un clone del árbol etiquetado del individual tree, y el conjunto *repeated* de vértices está vacío. A continuación, recorre el árbol clonado procesando los vértices de tipo concepto. El procesamiento consiste en calcular los vértices colapsables del vértice que se está estudiando. Para cada conjunto de vértices colapsables con más de un elemento, el algoritmo poda los subárboles asociados a todos los vértices colapsables excepto uno que es añadido al conjunto *repeated*. El algoritmo termina cuando el se ha recorrido totalmente el árbol etiquetado. La Figura §8.2 muestra una traza del algoritmo.

Configuraciones: Las configuraciones que necesitamos son tuplas compuestas por cuatro elementos:

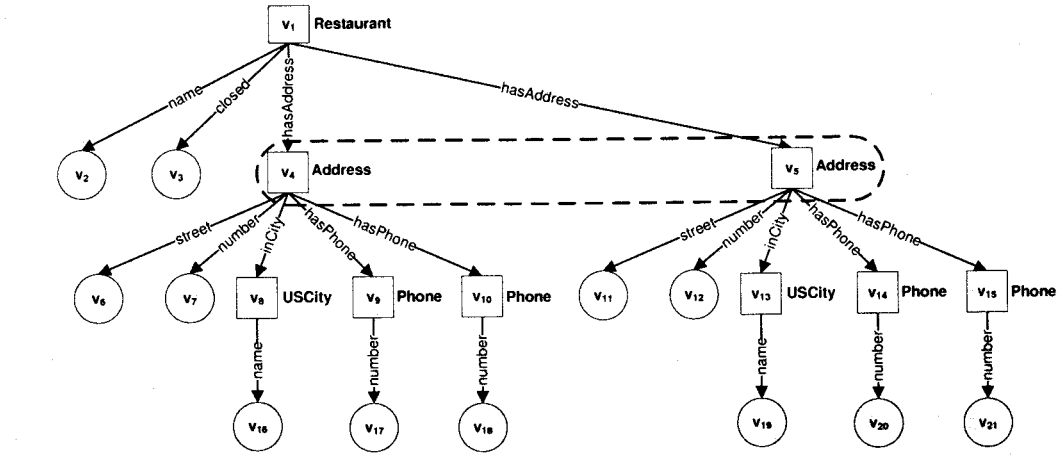
$$Config_{BSD} == IndividualTree \times LabelledTree \times \mathbb{P}\, Vertex \times \mathbb{P}\, Vertex$$

donde el primer elemento es un individual tree que cumple los requisitos exigidos, el segundo y tercer elemento componen la descripción semántica que queremos construir y el último elemento almacena los vértices a estudiar.

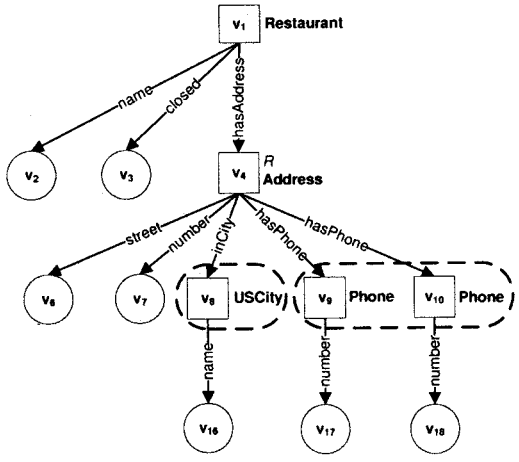
Predicados y funciones: Dos funciones son usadas para actualizar las configuraciones: *pruneSubtrees* y *partitionChildren*. Además, la función *cloneLt* es usada para definir la configuración inicial del algoritmo.

1. La función *pruneSubtrees* poda de un árbol etiquetado aquellos subárboles asociados a un conjunto de vértices.

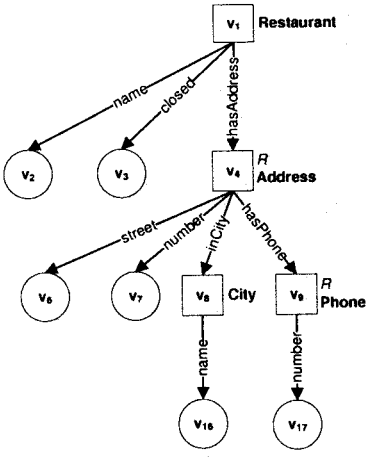
$$
\begin{array}{|l}
pruneSubtrees : LabelledTree \times \mathbb{P}\, Vertex \nrightarrow LabelledTree \\
\hline
\forall\, lt_1, lt_2 : LabelledTree;\ sv_1, sv_2 : \mathbb{P}\, Vertex \mid sv_1 \subseteq lt_1.t.vertices \bullet \\
\quad pruneSubtrees(lt_1, sv_1) = lt_2 \Leftrightarrow \\
\qquad sv_2 = \{p : paths(lt_1.t);\ v : sv_1;\ i,j : \mathbb{N} \mid v \in \operatorname{ran} p \land p(i) = v \land j \geq i \bullet p(j)\} \land \\
\qquad lt_2.t.vertices = lt_1.t.vertices \setminus sv_2 \land \\
\qquad lt_2.t.edges = \{(v_1, v_2) : lt_1.t.edges \bullet v_1 \notin sv_2 \land v_2 \notin sv_2 \bullet (v_1, v_2)\} \land \\
\qquad lt_2.vertexType = \{v : lt_2.t.vertices \bullet v \mapsto lt_1.vertexType(v)\} \land \\
\qquad lt_2.edgeType = \{e : lt_2.t.edges \bullet e \mapsto lt_1.edgeType(e)\}
\end{array}
$$



(a) Estudio del vértice v_1



(b) Estudio del vértice v_4



(c) descripción semántica devuelta

Figura 8.2: *Ejemplo de* `buildSD`.

2. La función *partitionChildren* particiona los hijos de tipo concepto de un vértice de tipo concepto.

$$partitionChildren : LabelledTree \times Vertex \rightarrowtail \mathbb{P}\,\mathrm{seq}\,Vertex$$

$$\begin{aligned}
&\forall lt : LabelledTree;\ v : Vertex;\ ssv : \mathbb{P}\,\mathrm{seq}\,Vertex \mid v \in lt.t.vertices \bullet \\
&\quad (notFillerChildren(lt, v) \neq \varnothing \bigwedge partitionChildren(lt, v) = ssv \Leftrightarrow \\
&\qquad \forall sv : ssv \bullet collapsableVertices(lt, \mathrm{ran}\, sv) \bigwedge \\
&\qquad \#notFillerChildren(lt, v) = \bigcup\{sv : ssv \bullet \mathrm{ran}\, sv\} \bigwedge \\
&\qquad \forall sv : ssv \bullet \#sv = \#\,\mathrm{ran}\, sv) \bigvee \\
&\quad (notFillerChildren(lt, v) = \varnothing \bigwedge partitionChildren(lt, v) = \varnothing)
\end{aligned}$$

$$notFillerChildren : LabelledTree \times Vertex \rightarrowtail \mathbb{P}\,Vertex$$

$$\begin{aligned}
&\forall lt : LabelledTree;\ v : lt.t.vertices \bullet notFillerChildren(lt, v) = \\
&\quad \{v_x : children(v) \mid lt.vertexType(v_x) \neq FILLER\}
\end{aligned}$$

3. La función *cloneLt* clona un árbol etiquetado.

$$cloneLt : LabelledTree \rightarrow LabelledTree$$

$$\begin{aligned}
&\forall lt_1, lt_2 : LabelledTree \bullet cloneLt(lt_1) = lt_2 \Leftrightarrow \\
&\quad \exists f : lt_1.t.vertices \rightarrowtail\!\!\!\!\rightarrow lt_2.t.vertices \bullet \\
&\qquad \forall (v_1, v_2) : lt_1.t.edges \bullet \\
&\qquad\quad (f(v_1), f(v_2)) \in lt_2.t.edges \bigwedge \\
&\qquad\quad lt_1.edgeType(v_1, v_2) = lt_2.edgeType(f(v_1), f(v_2)) \bigwedge \\
&\qquad \forall v : lt_1.t.vertices \bullet lt_1.vertexType(v) = lt_2.vertexType(f(v))
\end{aligned}$$

Reglas: Las reglas son definidas formalmente como una relación homogénea entre dos configuraciones: $\rightarrow_{BSD}$: $Config_{BSD} \leftrightarrow Config_{BSD}$. El algoritmo `buildSD` se define a partir de las siguientes reglas:

1. Si el vértice visitado tiene hijos de tipo concepto, usamos la función *partitionChildren* para particionarlos en vértices colapsables. Uno de los vértices de cada partición es añadido a *sv* (conjunto de vértices a estudiar); también, si la partición tiene más de un vértice, el vértice se añade al conjunto *rep* y se podan los subárboles asociados al resto de los vértices.

$$(it, lt, rep, sv) \rightarrow_{BSD} (it, lt', rep', sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ pc \neq \varnothing \end{array} \right]$$

$$\textit{Where} \quad \left\{ \begin{array}{l} v \triangleq member(sv) \\ pc \triangleq partitionChildren(lt, v) \\ sv' \triangleq (sv \setminus \{v\}) \cup \{s : pc \bullet head(s)\} \\ lt' \triangleq pruneSubtrees(lt, \bigcup\{s : pc \bullet \operatorname{ran} tail(s)\}) \\ rep' \triangleq rep \cup \{s : pc \mid \#s > 1 \bullet head(s)\} \end{array} \right.$$

2. Si el vértice visitado no tiene hijos de tipo concepto, actualizamos sv eliminando el vértice visitado.

$$(it, lt, rep, sv) \rightarrow_{BSD} (it, lt, rep, sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ pc = \varnothing \end{array} \right]$$

$$\textit{Where} \quad \left\{ \begin{array}{l} v \triangleq member(sv) \\ pc \triangleq partitionChildren(lt, v) \\ sv' \triangleq sv \setminus \{v\} \end{array} \right.$$

Algoritmo: El algoritmo viene especificado como el proceso de aplicar las reglas anteriores tantas veces como sean necesarias para partiendo de la configuración inicial definida como $(it, cloneLt(it.lt), \varnothing, \langle root(it.lt.t)\rangle)$ llegar a la configuración final $(it, lt, rep, \langle\,\rangle)$:

$$\texttt{buildSD} : IndividualTree \nrightarrow SemanticDescription$$

$$\begin{array}{l} \forall\, it : IndividualTree;\ sd : SemanticDescription;\ lt : LabelledTree;\ rep : \mathbb{P}\, Vertex \bullet \\ \quad \texttt{buildSD}(it) = sd \Leftrightarrow \\ \qquad (it, cloneLt(it.lt), \varnothing, \langle root(it.lt.t)\rangle) \xrightarrow{!}_{BSD} (it, lt, rep, \langle\,\rangle) \wedge \\ \qquad sd.lt = lt \wedge sd.repeated = rep \end{array}$$

8.2.2. Corrección

Teorema 8.1 (Terminación) *El algoritmo* `buildSD` *termina.*

Demostración Inicialmente, el árbol etiquetado de la descripción semántica es un clon del árbol etiquetado del individual tree. El algoritmo recorre el individual tree clonado y sólo procesa los vértices de tipo concepto. Los vértices a

procesar se almacenan en el conjunto *sv*. Una vez que un vértice de tipo concepto es estudiado se elimina de *sv*. Dado que los individual trees tienen un número finito de vértices y no tienen ciclos, el algoritmo `buildSD` termina. □

Teorema 8.2 (Corrección) *El algoritmo* `buildSD` *es correcto con respecto a la especificación de la función buildSD.*

Demostración La prueba consiste en dos partes: en primer lugar tenemos que probar que los vértices colapsables en el individual tree son colapsados en la descripción semántica (1); también es necesario probar que las rutas colapsables en el individual tree son colapsadas en la descripción semántica (2).

1. El algoritmo usa la función *partitionChildren* para obtener una partición de los hijos del vértice que está siendo estudiado, en secuencias de vértices colapsables. Por cada secuencia, sólo se almacena un vértice en el conjunto de vértices a estudiar y los subárboles del resto de los vértices en una misma partición son podados. Es decir, se colapsan los vértices colapsables. Dado que se estudian todos los vértices de tipo concepto en el árbol etiquetado de salida, la descripción semántica no tiene vértices colapsables.
2. Inmediato a partir del lema §7.1 y (1).

□

8.2.3. Complejidad

El algoritmo `buildSD` visita todos los vértices en el individual tree y por cada vértice de tipo concepto, particiona sus hijos de tipo concepto en vértices colapsables. Por tanto, la complejidad del algoritmo `buildSD` es $O(n \cdot b)$, donde n y b son el número total de vértices y anchura del individual tree, respectivamente.

8.3. Construcción de *Location*

El algoritmo toma a la entrada un Abox, un *HierarchicalSlot* y una descripción semántica global; de manera que el Abox representa semánticamente la información en el *HierarchicalSlot*, y la descripción semántica global contiene

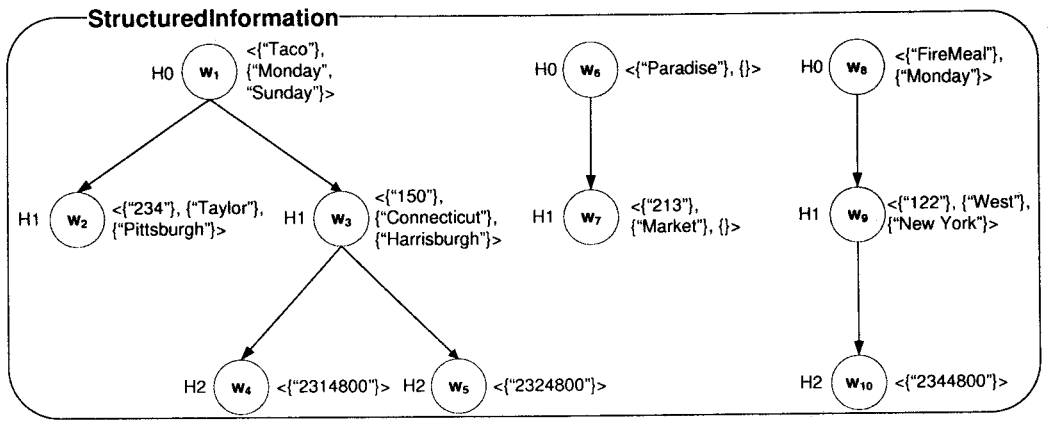


Figura 8.3: *StructuredInformation (repetición de la figura §6.2(b)).*

toda la información semántica necesaria para traducir todos los individuos del sitio web. El algoritmo devuelve la información de localización. Al igual que con el algoritmo `buildSD`, es necesario que el usuario proporcione un Abox y un *HierarchicalSlot* que cumplan las siguientes restricciones:

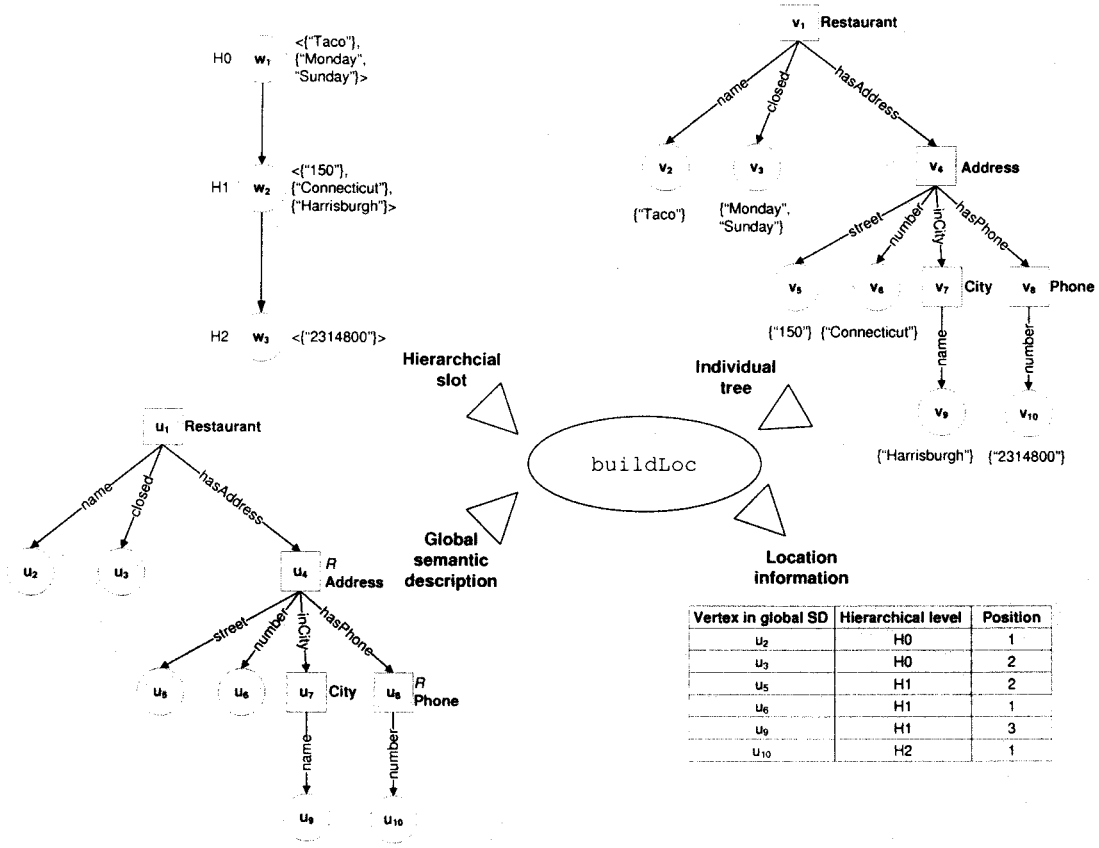
Requisito 1. Todas los nombres de conceptos y propiedades necesarias para dar semántica a la información en el sitio web aparecen en el Abox.

Requisito 2. Cada propiedad opcional o de cardinalidad múltiple, aparece una única vez.

Requisito 3. No existen atributos sin valor en el *HierarchicalSlot*, y todos los valores de los atributos son distintos.

Previos requisitos evitan problemas de ambigüedad en la obtención de la información de localización. De nuevo, el algoritmo se presenta en términos de individual trees en vez de Aboxes.

Ejemplo 8.2 *El individual tree de la Figura §8.1(a) no cumple ni el primero, ni el segundo de los requisitos anteriores. El restaurante RID_1 tiene más de una dirección, la dirección AID_2 tiene más de un número de teléfono, y la dirección AID_1 no tiene ningún número de teléfono. De la misma forma, el segundo HierarchicalSlot de la figura §8.3 no cumple el tercer requisito, ya que contiene atributos sin valor.*



Vertex in global SD	Hierarchical level	Position
u_2	H0	1
u_3	H0	2
u_5	H1	2
u_6	H1	1
u_9	H1	3
u_{10}	H2	1

Figura 8.4: *Ejemplo de* `buildLoc`.

8.3.1. Algoritmo

El algoritmo `buildLoc` obtiene la información de localización como sigue: recorre el árbol etiquetado de la descripción semántica procesando vértices de tipo *FILLER*. El procesamiento consiste en mirar por un vértice en el individual tree que sea el reflejado (único debido a los requisitos impuestos) del vértice que se está estudiando y buscando en el *HierarchicalSlot* su atributo. El algoritmo termina cuando el árbol etiquetado ha sido totalmente recorrido. La Figura §8.4 ilustra gráficamente la información de localización obtenida para una descripción semántica global, un individual tree y un *HierarchicalSlot*. Por ejemplo, el nivel jerárquico y posición para el vértice u_2 de la descripción semántica se obtiene buscando el atributo {"*Taco*"} en el *HierarchicalSlot*.

Configuraciones: Las configuraciones que necesitamos son tuplas compuestas por cinco elementos:

$Config_{BL} == IndividualTree \times HierarchicalSlot \times SemanticDescription \times Location \times \mathbb{P}\, Vertex$

donde el primer y segundo elemento en las tuplas se refiere a un individual tree y a un *hierarchicalSlot* que cumplen los requisitos exigidos; el tercer elemento es la descripción semántica global; el cuarto elemento es la información de localización que tenemos que calcular; el último elemento almacena los vértices a estudiar.

Predicados y funciones:

1. La función *sameVertex* busca un vértice en el individual tree que es el reflejado de un vértice en la descripción semántica.

$$sameVertex : Vertex \times LabeledTree \times LabeledTree \nrightarrow Vertex$$

$$\begin{aligned}&\forall v_1, v_2 : Vertex;\ lt_1, lt_2 : LabeledTree \mid v_1 \in lt_1 \bullet sameVertex(v_1, lt_1, lt_2) = v_2 \Leftrightarrow \\ &\quad \exists p_1 : paths(lt_1.t);\ p_2 : paths(lt_2.t);\ i : \mathbb{N} \mid \\ &\qquad (p_1, lt_1) \sim (p_2, lt_2) \land p_1(i) = v_1 \bullet p_2(i) = v_2\end{aligned}$$

2. La función *findAttribute* busca un atributo en un *HierarchicalSlot*.

$$findAttribute : HierarchicalSlot \times Attribute \nrightarrow \mathbb{N} \times Label$$

$$\begin{aligned}&\forall hs : HierarchicalSlot;\ a : Attribute;\ n : \mathbb{N};\ l : Label \bullet \\ &\quad findAttribute(hs, a) = (n, l) \Leftrightarrow \\ &\qquad \exists_1 v : hs.t.vertices;\ sa : \mathrm{seq}\, Attribute;\ a : Attribute \bullet \\ &\qquad\quad hs.vertexAttributes(v) = sa \land sa(n) = a \land hs.vertexHLevel(v) = l\end{aligned}$$

Reglas: Las reglas se definen como reaciones homogéneas entre dos configuraciones: $\rightarrow_{BL}$: $Config_{BL} \leftrightarrow Config_{BL}$. El algoritmo se define a partir de las siguientes reglas:

1. Si el vértice visitado v es de tipo *FILLER*, calculamos su información de localización y lo eliminamos del conjunto de vértices a estudiar (sv).

$$(it, hs, sd, loc, sv) \rightarrow_{BL} (it, hs, sd, loc', sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ sd.lt.vertexType(v) = FILLER \end{array}\right]$$

$$Where \begin{cases} v \triangleq member(sv) \\ a \triangleq it.vertexAttribute(sameVertex(v, sd.lt, it.lt)) \\ (pos, hlevel) \triangleq findAttribute(hs, a) \\ loc' \triangleq (\!|\ vertexPosition \rightsquigarrow loc.vertexPosition \cup \{v \mapsto hlevel\}, \\ \qquad\quad vertexLevel \rightsquigarrow loc.vertexLevel \cup \{v \mapsto hlevel\}\ |\!) \\ sv' \triangleq sv \setminus \{v\} \end{cases}$$

2. Si el vértice visitado v no es de tipo *FILLER*, entonces actualizamos sv eliminando v y añadiendo sus hijos.

$$(it, hs, sd, loc, sv) \rightarrow_{BL} (it, hs, sd, loc, sv') \quad \left[\begin{array}{l} sv \neq \varnothing \\ sd.lt.vertexType(v) \neq FILLER \end{array} \right]$$

$$\textit{Where} \quad \left\{ \begin{array}{l} v \triangleq member(sv) \\ sv' \triangleq (sv \setminus \{v\}) \cup children(sd.lt.t, v) \end{array} \right.$$

Algoritmo: el algoritmo viene especificado como el proceso de aplicar las reglas anteriores tantas veces como sean necesarias para partiendo de la configuración inicial definida como $(it, hs, sd, (\!| \textit{vertexPosition} \rightsquigarrow \varnothing, \textit{vertexLevel} \rightsquigarrow \varnothing |\!), \langle Root(sd.lt) \rangle)$ llegar a la configuración final $(it, hs, sd, loc, \langle\,\rangle)$:

$$\texttt{buildLoc} : \textit{IndividualTree} \times \textit{HierarchicalSlot} \times \textit{SemanticDescription} \nrightarrow \textit{Location}$$

$$\forall\, it : \textit{IndividualTree};\ hs : \textit{HierarchicalSlot};\ sd : \textit{SemanticDescription};\ loc : \textit{Location} \bullet$$
$$\texttt{buildPos}(it, hs, sd) = loc \Leftrightarrow$$
$$(it, hs, sd, (\!| \textit{vertexPosition} \rightsquigarrow \varnothing, \textit{vertexLevel} \rightsquigarrow \varnothing |\!), \langle root(sd.lt) \rangle) \xrightarrow{!}_{BL} (it, hs, sd, loc, \langle\,\rangle)$$

8.3.2. Corrección

Teorema 8.3 (Terminación) *El algorithm* `buildLoc` *termina.*

Demostración El algoritmo recorre todos los vértices del árbol etiquetado de la descripción semántica. Una vez un vértice es procesado, se elimina del conjunto de vértices a estudiar (sv). Los vértices de tipo concepto añaden sus hijos a sv. Dado que los individual trees tienen un número finito de vértices y no tienen ciclos, el algoritmo `buildSD` termina. □

Teorema 8.4 (Corrección) *El algorithm* `buildLoc` *es correcto respecto a la especificación de la función buidLoc.*

Demostración La prueba es inmediata a partir de los requisitos impuestos al inicio de esta sección. Por cada área de influencia en la descripción semántica existe una única área de influencia reflejada en el individual tree, además todos los atributos en el *HierarchicalSlot* son distintos, posibilitando el cálculo de las posiciones y de los niveles jerárquicos para todos los vértices de tipo *FILLER* en el árbol etiquetado de la descripción semántica. □

8.3.3. Complejidad

El algoritmo `buildLoc` visita todos los vértices en la descripción semántica, y por cada vértice de tipo *FILLER*, busca sus atributos en el *HierarchicalSlot*; la información de localización se asocia con el vértice reflejado en la descripción semántica. Por tanto, en el peor caso, la complejidad de `buildLoc` es $O(n \cdot m \cdot d)$, donde n es el número de vértices en el individual tree, m es el número de vértices en el *HerarchicalSlot*, y d es la profundidad del árbol etiquetado de la descripción semántica.

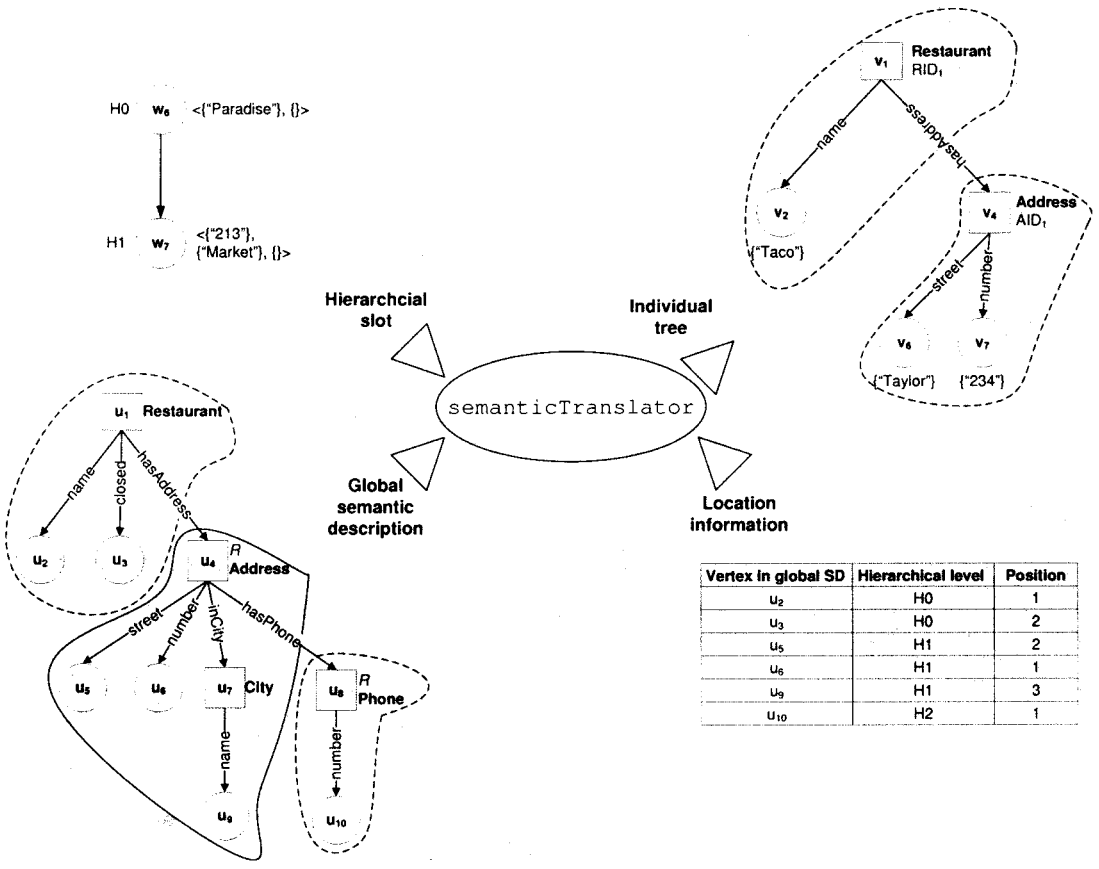
8.4. Traductor semántico

El algoritmo `semanticTranslator` toma a la entrada información estructurada (*StructuredInformation*), y usa una descripción semántica global y la información de localización, para construir un Abox que representa semánticamente esa información. El algoritmo se presenta en términos de traducción de un *HierarchicalSlot*.

8.4.1. Algoritmo

Está implementado como un algoritmo recurrente que trabaja basado en la idea de que cada vértice en el *HierarchicalSlot* tiene asociado un área de influencia reflejada, y que cada área de influencia reflejada se puede representar como un individual tree. La recursión se aplica sobre el *HierarchicalSlot*. La condición de parada es que el vértice del *HierarchicalSlot* estudiado no tenga hijos, en tal caso, se devuelve el individual tree asociado a su área de influencia reflejada. La llamada recurrente es que el vértice del *HierarchicalSlot* estudiado tenga hijos, en tal caso, el individual tree correspondiente al área de influencia reflejada del vértice padre se conecta a los individual trees obtenidos recursivamente para sus hijos.

Ejemplo 8.3 *La Figura §8.5 ilustra gráficamente el individual tree que se obtiene para la descripción semántica global, la información de localización, y un HierarchicalSlot. Observe que el individual tree tiene dos áreas de influencia reflejadas, que se han obtenido clonando las áreas de influencia representadas por los vértices u_1 y u_4 de la descripción semántica; también que todos los vértices y ejes en un área de influencia no tiene porque ser clonados, por ejemplo, el vértice u_3 no se ha clonado porque en el HierarchicalSlot no existe informacón acerca de que días cierra el restaurante; por último, no existen áreas de influencia reflejadas para el área de influencia representada*



Vertex in global SD	Hierarchical level	Position
u_2	H0	1
u_3	H0	2
u_5	H1	2
u_6	H1	1
u_9	H1	3
u_{10}	H2	1

Figura 8.5: *Ejemplo de* `sematicTranslator`.

por el vértice u_8 ya que el HierarchicalSlot no contiene información acerca de números de teléfono.

Configuraciones: tres tipos de configuraciones son necesarias para definir el algoritmo de traducción semántica. Su significado se mostrará despúes, cuando se presente el algoritmo.

1. $Config_{ST_1} == SemanticDescription \times HierarchicalSlot \times Location \times Vertex$
2. $Config_{ST_2} == IndividualTree$
3. $Config_{ST_3} == SemanticDescription \times HierarchicalSlot \times Location \times \mathbb{P}\, Vertex \times \mathbb{P}\, IndividualTree \times \text{seq}\, Label$

Predicados y funciones:

1. La Función *cloneInfluenceArea* toma a la entrada una descripción semántica, un *HierarchicalSlot*, la información de localización y un vértice del *HierarchicalSlot* y devuelve un individual tree. Obtiene el individual tree clonando un conjunto de nodos del árbol etiquetado de la descripción semántica y calculando las funciones *vertexIndividualName* y *vertexAttribute* para los vértices clonados. Los vértices a clonar son devueltos por la función *verticesToClone*. *vertexIndividualName* se calcula a partir de la función *getIdentifier*, y *vertexAttribute* se obtiene a partir de la información de localización y del *HierarchicalSlot*.

$$
\begin{array}{|l}
cloneInfluenceArea : SemanticDescription \times HierarchicalSlot \times Location \times \\
\quad Vertex \rightarrow IndividualTree \\
\hline
\forall\, sd : SemanticDescription;\ hs : HierarchicalSlot;\ loc : Location;\ w : Vertex; \\
\quad it : IndividualTree \bullet cloneInfluenceArea(sd, hs, loc, w) = it \Leftrightarrow \\
\qquad \exists f : verticesToClone(sd, hs, loc, hs.vertexHLevel(w), w) \rightarrowtail\!\!\!\!\!\!\rightarrow it.lt.t.vertices \bullet \\
\qquad\quad \forall (u_1, u_2) : sd.lt.t.edges \mid u_1 \in \mathrm{dom} f \land u_2 \in \mathrm{dom} f \bullet \\
\qquad\qquad (f(u_1), f(u_2)) \in it.lt.t.edges \land \\
\qquad\qquad sd.lt.edgeType(u_1, u_2) = it.lt.edgeType(f(u_1), f(u_2)) \land \\
\qquad\quad \forall u : \mathrm{dom} f \bullet \\
\qquad\qquad sd.lt.vertexType(u) = it.lt.vertexType(f(u)) \land \\
\qquad\qquad (sd.lt.vertexType(u) = FILLER \land \\
\qquad\qquad\quad it.vertexAttribute(f(u)) = \\
\qquad\qquad\qquad hs.vertexAttributes(w)(loc.vertexPosition(u)) \lor \\
\qquad\qquad sd.lt.vertexType(u) \neq FILLER \land \\
\qquad\qquad\quad it.vertexIndividualName(f(u)) = getIdentifier(it, f(u)))
\end{array}
$$

La Función *verticesToClone* devuelve un conjunto de vértices de la descripción semántica que pertenecen al área de influencia asociada a un vértice del *HierarchicalSlot*. Los vértices devueltos, o bien son vértices de tipo *FILLER* a los que la información de localización le asocia atributos no vacíos, o vértices de tipo concepto que pertenezca a una ruta que termine en un vértice de tipo *FILLER* al que le corresponda un atributo no vacío.

$$
\begin{array}{|l}
verticesToClone : SemanticDescription \times HierarchicalSlot \times Location \times Label \times \\
\quad Vertex \rightarrow \mathbb{P}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ hs : HierarchicalSlot;\ loc : Location;\ lb : Label; \\
\quad w : Vertex \bullet verticesToClone(sd, lb, w) = \\
\qquad \{v : influenceAreaLabel(sd, loc, lb) \mid sd.lt.vertexType(v) = FILLER \wedge \\
\qquad\quad hs.vertexAttributes(w)(l.vertexPosition(v)) \neq \varnothing\} \cup \\
\qquad \{v : influenceAreaLabel(sd, loc, lb) \mid sd.lt.vertexType(v) \neq FILLER \wedge \\
\qquad\quad (\exists\, p : paths(sd.lt.t);\ v_x : Vertex \mid v_x = p(\#p) \wedge v \in ran(p) \bullet \\
\qquad\qquad hs.vertexAttributes(w)(loc.vertexPosition(v_x)) \neq \varnothing)\}
\end{array}
$$

La Función *influenceAreaLabel* devuelve los vértices de la descripción semántica que pertenecen al área de influencia asociada a una determinada etiqueta.

$$
\begin{array}{|l}
influenceAreaLabel : SemanticDescription \times Location \times Label \rightarrow \mathbb{P}\, Vertex \\
\hline
\forall\, sd : SemanticDescription;\ loc : Location;\ lb : Label;\ su : \mathbb{P}\, Vertex \bullet \\
\quad influenceAreaLabel(sd, loc.lb) = su \Leftrightarrow \\
\qquad influenceArea(sd, su) \wedge \\
\qquad \exists\, u : su \mid sd.lt.vertexType(u) = FILLER \bullet loc.vertexLevel(u) = lb
\end{array}
$$

La Función *getIdentifier* usa un algoritmo de hashing para proporcionar un identificador único a los vértices de tipo concepto. El algoritmo de hashing toma a la entrada los valores de los vértices de tipo *FILLER* alcanzables desde el vértice de tipo concepto.

$$
\begin{array}{|l}
getIdentifier : IndividualTree \times Vertex \rightarrow IndividualName \\
\hline
\forall\, it : IndividualTree;\ v : Vertex \mid v \in it.lt.t.vertices \bullet getIdentifier(it, v) = \\
\quad hashing(\bigcup \{p : paths(it.lt.t) \mid v \in \operatorname{ran} p \bullet it.vertexAttribute(p(\#p))\})
\end{array}
$$

2. La Función *attach* toma como entrada una descripción semántica, la información de localización, un individual tree, una etiqueta, una secuencia de individual tree y una secuencia de etiquetas, y devuelve un nuevo individual tree. El individual tree representa un área de influencia reflejada (del área de influencia identificada por la etiqueta) obtenida de un vértice del *HierarchicalSlot;* la secuencia de individual tree representa áreas de influencia reflejadas (de las áreas de influencia identificadas por la secuencia de etiquetas) de los hijos del vértice anterior. La conexión se lleva a cabo añadiendo nuevos ejes que conectan la raíz del individual tree con todas las raízes de los individual tree de la secuencia. Los tipos de las propiedades de los nuevos ejes son obtenidos por la función *propertyType*.

$$
\begin{array}{|l}
attach : semanticDescription \times Location \times IndividualTree \times Label \times \\
\quad \text{seq}\, IndividualTree \times \text{seq}\, Label \rightarrow IndividualTree \\
\hline
\forall\, sd : semanticDescription;\ loc : Location;\ it, it_x : IndividualTree;\ lb : Label; \\
\quad sit : \text{seq}\, IndividualTree;\ slb : \text{seq}\, Label \bullet attach(sd, it, lb, sit, slb) = it_x \Leftrightarrow \\
\qquad it_x.lt.t.vertices = it.lt.t.vertices \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.t.vertices\} \\
\qquad it_x.lt.t.edges = it.lt.t.edges \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.t.edges\} \cup \\
\qquad\quad \{ita : \text{ran}\, sit \bullet (root(it.lt.t), root(ita.lt.t))\} \\
\qquad it_x.lt.vertexType = it.lt.vertexType \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.vertexType\} \\
\qquad it_x.lt.edgeType = it.lt.edgeType \cup \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.edgeType\} \cup \\
\qquad\quad \{i : 1 \,..\, \#sit \bullet (root(it.lt.t), root(ita.lt.t)) \mapsto propertyType(sd, loc, lb, slb(i))\} \\
\qquad it_x.lt.vertexIndividualName = it.lt.vertexIndividualName \cup \\
\qquad\quad \bigcup\{ita : \text{ran}\, sit \bullet ita.lt.vertexIndividualName\} \\
\qquad it_x.lt.vertexAttribute = it.lt.vertexAttribute(v) \cup \\
\qquad\quad \{ita : \text{ran}\, sit \bullet ita.lt.vertexAttribute\}
\end{array}
$$

La Función *propertyType* toma a la entrada una descripción semántica y dos etiquetas, y devuelve el nombre de la propiedad en la descripción semántica que conecta las áreas de influencia identificadas por esas etiquetas.

$$
\begin{array}{|l}
propertyType : SemanticDescription \times Location \times Label \times Label \rightarrow PropertyName \\
\hline
\forall\, sd : SemanticDescription;\ loc : Location;\ lb_1, lb_2 : Label;\ u_1, u_2 : Vertex \mid \\
\quad u_1 = subtreeRoot(influenceAreaLabel(sd, loc, lb_1)) \wedge \\
\quad u_2 = subtreeRoot(influenceAreaLabel(sd, loc, lb_2)) \bullet \\
\qquad propertyType(sd, lb_1, lb_2) = sd.lt.edgeType(u_1, u_2)
\end{array}
$$

Rules: el algoritmo se define a partir de las siguientes reglas, que son mutuamente recurrentes:

1. Para un vértice w del *HierarchicalSlot* con hijos, obtenemos el individual tree que representa su area de influencia reflejada, para ello clonamos de la descripción semántica el área de influencia asociada al nivel jerárquico al que w pertenece (*cloneInfluenceArea*); también, lo conectamos a los individual trees obtenidos para sus hijos (*attach*). Los individual tree para sus hijos (*sit*) y sus etiquetas (*slb*) se obtienen en el antecedente de la regla.

$$\frac{(sd, hs, loc, sw, \langle\,\rangle, \langle\,\rangle) \xrightarrow{!}_X (sd, hs, loc, \varnothing, sit, slb)}{(sd, hs, loc, w) \rightarrow_{ST} it} \quad [\ sw \neq \varnothing\ ]$$

$$\textit{Where} \begin{cases} sw \triangleq children(hs, w) \\ itc \triangleq cloneInfluenceArea(sd, hs, loc, w) \\ lb \triangleq hs.vertexHLevel(w) \\ it \triangleq attach(sd, loc, itc, lb, sit, slb) \end{cases}$$

2. Para cada hijo de un vértice en el *HierarchicalSlot*, se calcula de forma recurrente el individual tree que representa un área de influencia reflejada. Los individual tree son almacenados en una secuencia (*sit*), también las etiquetas asociadas a todos ellos (*slb*).

$$\frac{(sd, hs, loc, w) \rightarrow_{ST} it}{(sd, hs, loc, sw, sit, slb) \rightarrow_X (sd, hs, loc, sw', sit', slb')} \quad [\ sw \neq 0\ ]$$

$$\textit{Where} \begin{cases} sit' \triangleq sit \frown \langle it \rangle \\ slb' \triangleq slb \frown \langle hs.vertexHLevel(w) \rangle \\ w \triangleq member(sw) \\ sw' \triangleq sw \setminus \{w\} \end{cases}$$

3. Si w no tiene hijos, entonces el individual tree se obtiene clonando el área de influencia de una descripción semántica que está asociada al nivel jerárquico al que pertenece w.

$$(sd, hs, loc, w) \rightarrow_{ST} it \quad [\ sw = \varnothing\ ]$$

$$\textit{Where} \begin{cases} sw \triangleq children(hs, w) \\ it \triangleq cloneInfluenceArea(sd, HS, loc, w) \end{cases}$$

Algoritmo: El algoritmo viene especificado como el proceso de aplicar la regla $\rightarrow_{ST}$ para cada *HierarchicalSlot* en una *StructuredInformation*. La configuración inicial se define como $(sd, hs, loc, root(hs.t))$. El algoritmo viene especificado como sigue:

$$\texttt{semanticTranslator} : StructuredInformation \times SemanticDescription \times Location \rightarrow Abox$$

$$\forall\, si : StructuredInformation;\ sd : SemanticDescription;\ loc : Location \bullet$$
$$\texttt{semanticTranslator}(si, sd, loc) =$$
$$\bigcup \{hs : si;\ it : IndividualTree \mid (sd, hs, loc, root(hs.t)) \rightarrow_{ST} it \bullet toAbox(it)\}$$

8.4.2. Corrección

Teorema 8.5 (Terminación) *El algoritmo* `semanticTranslator` *termina.*

Demostración El algoritmo `semanticTranslator` recorre el *HierarchicalSlot*, comenzando por la raíz y recurrentemente estudiando los hijos; cada vez que el algoritmo es llamado para un vértice sin hijos, termina la recursión. Dado que los *HierarchicalSlot* tienen un número finito de vértices y no tienen ciclos, el algoritmo termina. □

Teorema 8.6 (Corrección) *El algoritmo* `semanticTranslator` *es correcto con respecto a la especificación de la función semanticTranslator.*

Demostración La prueba es inmediata atendiendo a la construcción del algoritmo. De acuerdo con el predicado *translation*, cada vértice en el *HierarchicalSlot* tiene asociado una área de influencia reflejada. Los vértices de tipo *FILLER* en el individual tree toman sus atributos de la información de localización de su vértice reflejado en la descripción semántica. También, los ejes del *HierarchicalSlot* tienen asociados los ejes del individual tree que conectan vértices que definen áreas de influencia reflejadas. □

8.4.3. Complejidad

El algoritmo calcula para cada vértice con hijos en el *HierarchicalSlot* su área de influencia asociada (suponemos que le lleva un tiempo constante *tc*) y lo conecta a las áreas obtenidas para sus hijos (suponemos que le lleva un tiempo constante *ta*). Si el vértice no tiene hijos, devuelve el correspondiente área de influencia reflejada (*tc*). En el peor caso, el *HierarchicalSlot* es un árbol k-ario balanceado (es necesario realizar el mismo número de llamadas recurrentes para cada vértice). Por tanto, la función temporal para la complejidad del algoritmo se define como:

$$T(m) = \begin{cases} k \cdot T(\frac{m}{k}) + tc + ta & \texttt{if } m > 1 \\ tc & \texttt{if } m = 1 \end{cases}$$

donde m es el número de vértices del *HierarchicalSlot* y k es la aridad del árbol.

Para resolver la ecuación anterior, calculamos la contribución en tiempo de los vértices a una misma profundidad en el árbol. A profundidad i el número

de vértices en el *HierarchicalSlot* es k^i. LA contribución en tiempo de cada vértice es $\frac{m}{k^i} + tc + ta$; por tanto, la contribución total a profundidad i es $(\frac{m}{k^i} + tc + ta) \cdot k^i$. El árbol tiene profundidad $\lg_k m$. Si definimos $c = tc + ta$, entonces:

$$T(m) = \sum_{i=0}^{\lg_k m} (m + c \cdot k^i) = \sum_{i=0}^{\lg_k m} m + c \cdot \sum_{i=0}^{\lg_k m} k^i = m \cdot \lg_k m + c \cdot (\frac{k^{\lg_k m + 1} - 1}{k - 1})$$

Por lo tanto, la complejidad del algoritmo `semanticTranslator` viene aproximada, en el peor de los casos, por $O(m \cdot \lg_k m + k^{\lg_k m}) = O(m \cdot \lg_k m + m) = O(m \cdot \lg m)$.

Parte IV

Notas finales

Capítulo 9

Conclusiones y trabajo futuro

When people agree, it is only in their conclusions; their reasons are always different.

Jorge A.N. de Santayana, 1863–1952
Spanish philosopher

El increíble éxito de Internet ha propiciado la investigación en tecnologías que tienen como objetivo mejorar la interacción de los humanos con la web. Desgraciadamente, la información que un usuario humano puede fácilmente interpretar es generalmente difícil de extraer e interpretar por los agentes software. Esta es la razón por la qué tales mejoras se ven generalmente como problemas desde el punto de vista del programador de agentes. La web semántica facilitará la tarea de extracción de información con significado, independientemente de la manera en la que dicha información sea presentada. Sin embargo, ésta no parece que se adopte en un futuro inmediato, lo que justifica de la necesidad de disponer de soluciones para resolver el problema mientras tanto.

El objetivo de esta tesis era dar soporte a la idea de que la información que reside en las páginas web no son entendidas por los agentes software a un coste razonable. En el cuerpo de esta memoria, motivamos esta idea, y describimos los problemas que aparecen cuando un agente software intenta acceder al conocimiento en la web. Estos problema eran debido al hecho que la web actual está orientada a presentar información a seres humanos, es dinámica, enorme y distribuida (véase el capítulo §5 para la descripción de estos problemas y la prueba de que los sistemas existentes no los solucionan). WebMeaning es nuestro enfoque para extraer información con significado de la web sintáctica actual. Sus ventajas principales son que asocia semántica con la información extraída, mejorando la interoperabilidad del agente

(véase el capítulo §7 para nuestra propuesta para la traducción semántica); trata los cambios en la estructura de una página web, mejorando la adaptabilidad (véase el capítulo §6 para la definición de los verificadores sintácticos) y consigue una separación completa de la funcionalidad necesaria para la extracción de conocimiento, automatizando el desarrollo de extractores de conocimiento distribuidos (véase el capítulo §8 donde se definen los algoritmos automáticos).

Los resultados de esta memoria no tienen que ser vistos como el final de un camino, sino como fuente de motivación para trabajos de investigación futuros en esta línea. Entre los muchos temas que quedan abiertos o pueden ser mejorados, pensamos que puede ser interesante la extensión del marco de trabajo para posibilitar el acceso a fuentes heterogéneas de información, por ejemplo, bases de datos federadas o bases de conocimiento. Conseguir acceso a bases de datos federadas implica la traducción del modelo relacional a una ontología, pero en un ambiente federado podrían existir muchas ontologías para el mismo dominio. Para lograr la interoperabilidad semántica, las diferentes ontologías deben ser capaces de interoperar o integrarse. Conseguir acceso a bases de conocimiento implica definir esquemas de traducción entre ontologías, apareciendo el mismo problema: la interoperabilidad semántica.

Parte V

Apéndices

Apéndice A

Notación

A.1. Z

El lenguaje de especificación formal Z está basado en la teoría de conjuntos y en la lógica de predicados. Extiende el uso de esos lenguajes, permitiendo esquemas. Los esquemas en Z tiene dos partes: una parte declarativa, donde se declaran variables y sus tipos, y una parte de predicados, donde se relacionan y restringen las variables. El tipo de un esquema puede ser considerado el producto Cartesiano de los tipos de cada unas de sus variables, sin ninguna noción de orden, pero con la restricción impuestas por los predicados. Se potencia la modularidad permitiendo que los esquemas introduzcan a su vez otros esquemas, la forma de seleccionar una variable en estos casos es escribiendo *schema.var*.

Para introducir un nuevo tipo en Z, del que queremos abstraernos de los elementos que los compone, se usa la notación "given set". Por ejemplo, con [*Vertex*] representamos el conjunto de todos los vértices. Si queremos indicar que una variable toma valor de un conjunto de valores o de un par ordenado de valores, escribimos $x : \mathbb{P}\, Vertex$ y $x : Vertex \times Vertex$, respectivamente.

La Tabla §A.1 resume la notación usada en esta memoria. Para un tratamiento más completo de Z, referimos al lector a uno de los numerosos textos existentes, tales como [18, 48, 74].

A.2. Método de Plotkin

Hemos utilizado el popular método de Plotkin para definir nuestros algoritmos [73] ya que es simple y potente. Se basa en reglas de inferencia de la

Notation	Description
	Definitions and declarations
$a : A$	Declarations
$A == B$	Abbreviated definition
	Logic
$p \land q$	Logical conjunction
$p \lor q$	Logical disjunction
$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existencial quantification
	Sets
$x \in A$	Set membership
$\varnothing$	Empty set
$A \subseteq B$	Set inclusion
$\{x, y, \ldots\}$	Set of elements
(x, y)	Ordered pair
$A \times B$	Cartesian product
$\mathbb{P} A$	Power set
$A \cap B$	Set intersection
$A \cup B$	Set union
$\bigcup A$	Generalised union
$\#A$	Size of a finite set
$\{a : A \mid P(a) \bullet f(a)\}$	Set comprehension
	Relations and functions
$\mathrm{dom}\, R$	Domain of a relation
$\mathrm{ran}\, R$	Range of a relation
$A \pfun B$	Partial function
$A \fun B$	Total function
$A \rightarrowtail B$	Total bijection

Cuadro A.1: *Resumen de la notación usada en la memoria.*

forma:

$$\frac{\textit{Antecedent}}{\textit{Consequent}} \left[\begin{array}{l} \textit{Applicability} \\ \textit{conditions} \end{array} \right] \qquad \textit{Consequent} \left[\begin{array}{l} \textit{Applicability} \\ \textit{conditions} \end{array} \right]$$

$$\textit{Where} \left\{ \textit{Definitions} \right. \qquad \textit{Where} \left\{ \textit{Definitions} \right.$$

Para definir un algoritmo utilizando este método, es necesario identificar los datos sobre los que se trabaja y modelarlos como una tupla a la que habitualmente nos referimos como configuración del algoritmo.

Por ejemplo, para modelar el sistema del productor/consumidor, necesitaremos configuraciones de la forma(p, c, τ), donde p denota el estado del productor, c el estado del consumidor y τ la cola de tamaño fijo que almacena los elementos que están listos para ser consumidos. Obviamente, también necesitamos un par de reglas para referidas como $\rightarrow_{PROD}$ y $\rightarrow_{CONS}$ para describir como el productor y el consumidor cambian sus estados respectivamente. Estas reglas pueden dejarse sin especificar ya que al nivel de abstracción que nos movemos no es necesario profundizar en su semántica. Utilizando esta información, el sistema puede describirse con las siguientes reglas:

1. La regla de producción, que controla cómo los nuevos elementos son introducimos en la cola cuando hay espacio en ella. (Suponemos que MAX denota el número máximo de elementos que se pueden almacenar en τ.)

$$\frac{p \xrightarrow{i}_{PROD} p'}{(p, c, \tau) \rightarrow_{PC} (p', c', \tau')} \quad [\ |\tau| < MAX\ ]$$

$$\textit{Where} \left\{ \begin{array}{l} c' \triangleq c \\ \tau' \triangleq \textit{enqueue}(\tau, i) \end{array} \right.$$

$p \xrightarrow{i}_{PROD} p'$ significa que el productor produce un elemento i y que esto provoca una transición del estado p al estado p'. Fíjese que la regla puede aplicarse cuando $|\ \tau\ | < MAX$; de lo contrario, el productor tiene que esperar hasta que el consumidor elimine algún elemento de la cola.

2. La regla de consumición, que controla cómo los elementos son eliminados de la cola.

$$\frac{c \xrightarrow{i}_{CONS} c'}{(p,c,\tau) \rightarrow_{PC} (p',c',\tau')} \quad [\ |\tau| > 0 \land i = Head(\tau)\ ]$$

$$\textit{Where} \quad \begin{cases} p' \triangleq p \\ \tau' \triangleq dequeue(\tau) \end{cases}$$

3. Una regla que modela un ladrón, que roba un elemento de la cola. Esta regla no tiene antecedente.

$$(p,c,\tau) \rightarrow_{BURGLAR} (p',c',\tau') \quad [\ |\tau| > 0\ ]$$

$$\textit{Where} \quad \begin{cases} p' \triangleq p \\ \tau' \triangleq dequeue(\tau) \end{cases}$$

Las reglas de inferencia son de muy alto nivel, y permiten modelar algoritmos concurrentes con precisión, ya que pueden ser vistas como las descripción de las acciones atómicas que pueden ser ejecutadas en el sistema concurrente. El comportamiento o el algoritmo puede ser visto como el conjunto de todos los posibles entrelazados permitidos por las reglas de inferencia usadas para describirlo.

A.3. Tipo de datos árbol

El tipo de datos *Tree* especifica un árbol dirigido acíclico como un conjunto de vértices y ejes.

$[Vertex]$
$Edge == Vertex \times Vertex$

Tree

$vertices : \mathbb{P}\, Vertex$
$edges : \mathbb{P}\, Edge$

$\forall (v_1, v_2) : edges \bullet v_1 \in vertices \land v_2 \in vertices$
$\forall p : paths(\{\!|\ vertices \rightsquigarrow vertices, edges \rightsquigarrow edges\ |\!\}); i,j : \mathbb{N} \mid i,j : 1\,..\,(\#p) \land i \neq j \bullet$
$p(i) \neq p(j)$
$\forall v : vertices \bullet \exists p : paths(\{\!|\ vertices \rightsquigarrow vertices, edges \rightsquigarrow edges\ |\!\}) \bullet v \in \operatorname{ran} p$

A continuación, definimos un predicado y algunas funciones sobre el tipo de datos *Tree* que son muy usadas en esta memoria:

1. El predicado *isLeaf* se satisface si un vértice es hoja.

$$\begin{array}{|l} isLeaf : Tree \times Vertex \\ \hline \forall\, t : Tree;\ v : t.vertices \bullet isLeaf(t, v) \Leftrightarrow \nexists v_x : t.vertices \bullet (v, v_x) \in t.edges \end{array}$$

2. La función *root* devuelve la raíz de un árbol.

$$\begin{array}{|l} root : Tree \rightarrow Vertex \\ \hline \forall\, t : Tree;\ v : Vertex \bullet root(t) = v \Leftrightarrow \nexists v_x \bullet (v_x, v) \in t.edges \end{array}$$

3. La function *children* devuelve los hijos de un vértice.

$$\begin{array}{|l} children : Tree \times Vertex \nrightarrow \mathbb{P}\, Vertex \\ \hline \forall\, t : Tree;\ v : t.vertices \bullet children(t, v) = \{v_x : Vertex \mid (v, v_x) \in t.edges \bullet v_x\} \end{array}$$

4. La función *paths* devuelve todas las rutas del árbol.

$$\begin{array}{|l} paths : Tree \rightarrow \mathbb{P}\, \mathrm{seq}\, Vertex \\ \hline \forall\, t : Tree \bullet paths(t) = \\ \quad \{p : \mathrm{seq}\, Vertex \mid p(1) = root(t) \land isLeaf(t, p(\#p)) \land \\ \qquad (\forall\, i : 1\,..\,\#p - 1 \bullet (p(i), p(i+1)) \in t.edges) \bullet p\} \end{array}$$

5. La función *subtreeRoot* devuelve la raíz de un subárbol (especificado como un conjunto de vértices).

$$\begin{array}{|l} subtreeRoot : Tree \times \mathbb{P}\, Vertex \rightarrow Vertex \\ \hline \forall\, t : Tree;\ sv : \mathbb{P}\, Vertex;\ v : Vertex \bullet subtreeRoot(t, sv) = v \Leftrightarrow \\ \quad \forall\, v_x : sv \setminus \{v\} \bullet \exists\, p : paths(t);\ i, j : \mathbb{N} \bullet p(i) = v \land p(j) = v_x \land j > i \end{array}$$

Apéndice B

Equivalencia entre Abox y IndividualTree

Antes de probar la equivalencia entre *Abox* y *IndividualTree*, introduciremos un par de funciones de ayuda que nos permitirán diferenciar las aserciones de propiedades que establecen una relación entre dos conceptos, a aquellas que dan valores a los atributos asociados a los conceptos. La Función *getRelations* toma un *Abox* a la entrada y devuelve las aserciones de propiedades (tuplas en *propertyAssertions*) que tienen un nombre de individuo en la tercera posición. Por el contrario, la Función *getAttributes* devuelve aquellas que tienen literales en la tercera posición.

$$
\begin{array}{|l}
getRelations : Abox \rightarrow \mathbb{P}(PropertyName \times IndividualName \times IndividualName) \\
\hline
\forall a : Abox \bullet getRelations(a) = \\
\quad \{x : a.pn;\ y, z : a.IndividualNames \mid (x, y, z) \in a.propertyAssertions \bullet (x, y, z)\}
\end{array}
$$

$$
\begin{array}{|l}
getAttributes : Abox \rightarrow \mathbb{P}(PropertyName \times IndividualName \times Literal) \\
\hline
\forall a : Abox \bullet getAttributes(a) = \\
\quad \{x : a.pn;\ y : a.IndividualNames;\ l : Literal \mid (x, y, l) \in a.propertyAssertions \bullet (x, y, l)\}
\end{array}
$$

Observe que si a es un *Abox*, entonces $getRelations(a)$ y $getAttributes(a)$ definen una partición del conjunto $a.propertyAssertions$.

Ejemplo B.1 *Las salidas de getRelations y getAttributes para el Abox del ejemplo §6.2, son las siguientes:*

$$\begin{aligned}&\textbf{\textit{getRelations}}(a) = \{(hasAddress, RID_1, AID_1), (inCity, AID_1, CID_1),\\ &\quad (hasAddress, RID_1, AID_2), (inCity, AID_2, CID_2), (hasPhone, AID_2, PID_1),\\ &\quad (hasPhone, AID_2, PID_2)\}\end{aligned}$$

$$\begin{aligned}&\textbf{\textit{getAttributes}}(a) = \{(name, RID_1, \text{"Taco"}), (closed, RID_1, \text{"Monday"}),\\ &\quad (closed, RID_1, \text{"Sunday"}), (street, AID_1, \text{"Taylor"}), (number, AID_1, \text{"234"}),\\ &\quad (name, CID_1, \text{"Pittsburgh"}), (street, AID_2, \text{"Connecticut"}), (number, AID_2, \text{"150"}),\\ &\quad (name, CID_2, \text{"Harrisburgh"}), (number, PID_1, \text{"2314800"}), (number, PID_2, \text{"2324800"})\}\end{aligned}$$

B.1. Construcción de un *IndividualTree* a partir de un *Abox*

Teorema B.1 *Dado un Abox con aserciones sobre un único individuo, existe un único IndividualTree que representa a ese individuo.*

Demostración Para probar este teorema seguimos una aproximación constructiva que consiste en especificar un algoritmo abstracto que tome a la entrada $a : Abox$ y devuelva $it : IndividualTree$. El algoritmo es el siguiente:

1. Para cada aserción de concepto en el Abox, existe un vértice en el árbol:

$$\begin{aligned}&\forall (c, id) : a.conceptAssertions \bullet\\ &\quad \exists_1 v : Vertex \bullet\\ &\qquad v \in it.lt.t.vertices \bigwedge\\ &\qquad it.lt.vertexType(v) = c \bigwedge\\ &\qquad it.vertexIndividualName(v) = id\end{aligned}$$

2. Para cada aserción de propiedad (p, id_1, id_2) en $getRelations(a)$, existe un eje en el árbol que conecta los vértices identificados por id_1 e id_2 obtenidos en el paso (1):

$$\begin{aligned}&\forall (p, id_1, id_2) : getRelations(a) \bullet\\ &\quad \exists_1 v_1, v_2 : Vertex \bullet\\ &\qquad (v_1, v_2) \in it.lt.t.edges \bigwedge\\ &\qquad it.lt.edgeType(v_1, v_2) = p \bigwedge\\ &\qquad it.vertexIndividualName(v_1) = id_1 \bigwedge\\ &\qquad it.vertexIndividualName(v_2) = id_2\end{aligned}$$

3. Para cada conjunto de aserciones de propiedades en $getAttributes(a)$ con un mismo nombre de propiedad e individuo (id), existe un eje en el árbol que relaciona el vértice obtenido en el paso (1) para el individuo id con un nuevo vértice de tipo *FILLER*:

$$
\begin{array}{l}
\forall\, p : PropertyName;\ id : IndividualName;\ att : Attribute \mid \\
\quad att = \{(p, id, l) : getAttributes(a) \bullet l\} \land att \neq \varnothing \bullet \\
\quad\quad \exists_1\, v_1, v_2 : Vertex \bullet \\
\quad\quad\quad (v_1, v_2) \in it.lt.t.edges \land \\
\quad\quad\quad it.lt.edgeType(v_1, v_2) = p \land \\
\quad\quad\quad it.vertexIndividualName(v_1) = id \land \\
\quad\quad\quad it.lt.vertexType(v_2) = FILLER \land \\
\quad\quad\quad it.vertexAttribute(v_2) = att
\end{array}
$$

La función *toIndividualTree* devuelve el individual tree para un Abox como se ha indicado en los pasos anteriores:

$$
\begin{array}{l}
toIndividualTree : Abox \rightarrow IndividualTree \\
\hline
\forall\, a : Abox;\ it : IndividualTree \bullet toIndividualTree(a) = it \Leftrightarrow \\
\quad \forall (c, id) : a.conceptAssertions \bullet \qquad [i] \\
\quad\quad \exists_1\, v : Vertex \bullet \\
\quad\quad\quad v \in it.lt.t.vertices \land \\
\quad\quad\quad it.lt.vertexType(v) = c \land \\
\quad\quad\quad it.vertexIndividualName(v) = id \land \\
\quad \forall (p, id_1, id_2) : getRelations(a) \bullet \qquad [ii] \\
\quad\quad \exists_1\, v_1, v_2 : Vertex \bullet \\
\quad\quad\quad (v_1, v_2) \in it.lt.t.edges \land \\
\quad\quad\quad it.lt.edgeType(v_1, v_2) = p \land \\
\quad\quad\quad it.vertexIndividualName(v_1) = id_1 \land \\
\quad\quad\quad it.vertexIndividualName(v_2) = id_2 \land \\
\quad \forall\, p : PropertyName;\ id : IndividualName;\ att : Attribute \mid \qquad [iii] \\
\quad\quad att = \{(p, id, l) : getAttributes(a) \bullet l\} \land att \neq \varnothing \bullet \\
\quad\quad\quad \exists_1\, v_1, v_2 : Vertex \bullet \\
\quad\quad\quad\quad (v_1, v_2) \in it.lt.edges \land \\
\quad\quad\quad\quad it.lt.edgeType(v_1, v_2) = p \land \\
\quad\quad\quad\quad it.vertexIndividualName(v_1) = id \land \\
\quad\quad\quad\quad it.lt.vertexType(v_2) = FILLER \land \\
\quad\quad\quad\quad it.vertexAttribute(v_2) = att \land \\
\quad (\#it.lt.t.Edges = \#getRelations(a) + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}) \land \qquad [iv] \\
\quad (\#it.lt.t.Vertices = \#a.conceptAssertions + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}
\end{array}
$$

Observe que se han añadido dos nuevas restricciones (iv): la primera indica que el número de ejes en el árbol es igual a la suma de aserciones obtenidas en el paso (2) y las obtenidas en el paso (3); y la otra indica que el número de vértices en el árbol es igual a la suma de aserciones de conceptos en el *Abox* con elt'número de vértices de tipo *FILLER* obtenido en el paso (3). Estas restricciones aseguran que el *IndividualTree* obtenido es único (No existe otro *IndividualTree* distinto que represente a ese *Abox*.

□

B.2. Construcción de un *IndividualTree* a partir de un *Abox*

Teorema B.2 *Dado un IndividualTree, existe un único Abox con aserciones sobre el individuo representado en el IndividualTree.*

Demostración

Sea it : *IndividualTree*, entonces su correspondiente a : *Abox* se construye de la siguiente manera:

1. Para cada vértice de tipo concepto en el árbol, existe una aserción de concepto en el Abox.

$$\begin{array}{l}\forall v : it.lt.t.vertices \bullet \\ \quad \exists_1 (c, id) : a.conceptAssertions \bullet \\ \quad\quad c = it.lt.vertexType(v) \bigwedge \\ \quad\quad id = it.vertexIndividualName(v)\end{array}$$

2. Para cada eje (v_1, v_2) en el que v_2 no es un vértice de tipo *FILLER*, existe una aserción de propiedad en a entre dos conceptos (los obtenidos en el paso (1) para los vértices v_1 y v_2).

$$\begin{array}{l}\forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) \neq FILLER \bullet \\ \quad \exists_1 (p, id_1, id_2) : a.propertyAssertions \bullet \\ \quad\quad p = it.lt.edgeType(v_1, v_2) \bigwedge \\ \quad\quad id_1 = it.vertexIndividualName(v_1) \bigwedge \\ \quad\quad id_2 = it.vertexIndividualName(v_2)\end{array}$$

3. Para cada eje (v_1, v_2) en el que v_2 es un vértice de tipo *FILLER*, el número de aserciones de propiedades que existen en a es igual al número de literales del atributo asociado a v_2. Esas propiedades relacionan una misma instancia de concepto (la obtenida para v_1 en el paso (i)) con los distintos literales del atributo)

$$\begin{array}{l}\forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) = FILLER \bullet \\ \quad \exists (p, id, l) : a.propertyAssertions \bullet \\ \quad\quad p = it.lt.edgeType(v_1, v_2) \wedge \\ \quad\quad id = it.vertexIndividualName(v_1) \wedge \\ \quad\quad l \in it.vertexAttribute(v_2)\end{array}$$

La función *toAbox* devuelve el *Abox* para un *IndividualTree* como se mostrado en los pasos anteriores:

$$\begin{array}{|l}\hline toAbox : IndividualTree \rightarrow Abox \\ \hline \forall it : IndividualTree;\ a : Abox \bullet toAbox(it) = a \Leftrightarrow \\ \quad \forall v : it.lt.t.Vertices \bullet \qquad [i] \\ \quad\quad \exists_1 (c, id) : a.ConceptAssertions \bullet \\ \quad\quad\quad c = it.lt.vertexType(v) \wedge \\ \quad\quad\quad id = it.vertexIndividualName(v) \wedge \\ \quad \forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) \neq FILLER \bullet \qquad [ii] \\ \quad\quad \exists_1 (p, id_1, id_2) : a.propertyAssertions \bullet \\ \quad\quad\quad p = it.lt.edgeType(v_1, v_2) \wedge \\ \quad\quad\quad id_1 = it.vertexIndividualName(v_1) \wedge \\ \quad\quad\quad id_2 = it.vertexIndividualName(v_2) \wedge \\ \quad \forall (v_1, v_2) : it.lt.t.edges \mid it.lt.vertexType(v_2) = FILLER \bullet \qquad [iii] \\ \quad\quad \exists (p, id, l) : a.propertyAssertions \bullet \\ \quad\quad\quad p = it.lt.edgeType(v_1, v_2) \wedge \\ \quad\quad\quad id = it.vertexIndividualName(v_1) \wedge \\ \quad\quad\quad l \in it.vertexAttribute(v_2) \wedge \\ \quad (\#it.lt.edges = \#getRelations(a) + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\}) \wedge \qquad [iv] \\ \quad (\#it.lt.vertices = \#a.ConceptAssertions + \#\{(p, id, l) : getAttributes(a) \bullet (p, id)\} \\ \hline\end{array}$$

De nuevo, dos nuevas restricciones (iv) se han añadido para asegurar que el *IndividualTree* obtenido es único. □

Apéndice C

Acrónimos

ASCII. American Standard Code for Information Interchange.

BPEL4WS. Business Process Execution Language for Web Services.

CREAM. CREAtion of Metadata.

DAML. DARPA agent markup language.

DAML+OIL. DARPA Agent Markup Language plus OIL.

DARPA. Defense Advanced Research Projects Agency.

DTD. Document Type Definition.

HTML. Hypertext Markup Language.

HTTP. Hypertext Transfer Protocol.

INTHELEX. INcremental THEory Learner from EXamples.

KIF. Knowledge Interchange Format.

OIL. Ontology Interchange Language.

OML. Ontology Markup Language.

OWL. Web Ontology Language.

RACER. Renamed ABox and Concept Expression Reasoner.

RAPIER. Robust Automated Production of Information Extraction Rules.

RDF. Resource Description Framework.

RDF-S. Resource Description Framework Schema.

SGML. Standard Generalised Markup Language.

SHOE. Simple HTML Ontology Extensions.

SOAP. Simple Object Access Protocol.

SPEM. Software Process Engineering Metamodel.

SQL. Structured Query Language.

SRV. Sequence Rules with Validation.

SWWS. Semantic Web Enabled Web Services.

UDDI. Universal Description, Discovery and Integration.

UML. Unified Modeling Language.

URI. The Uniform Resource Identifier.

W3C. World Wide Web Consortium.

WebOntEx. Web Ontology Extraction.

WIEN. Wrapper Induction ENvironment.

WSDL. Web Service Definition Language.

XML. Extensible Markup Language.

XOL. Ontology Exchange Language.

Bibliografía

[1] H. Alani, S. Kim, D.E. Millard, M.J. Weal, W. Hall, P.H. Lewis, and N.R. Shadbolt. Automatic ontology-based knowledge extraction from web documents. *IEEE Intelligent Systems*, 18(1):14–21, 2003.

[2] A.Y. and D.S. Weld. Intelligent internet systems. *Artificial Intelligence*, 118 (12):1–14, 2000.

[3] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The description logic handbook: theory, implementation and applications*. Cambridge University Press, 2003.

[4] T. Berners-Lee. WWW: past, present, and future. *Computer*, 29(10):69–77, 1996.

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[6] B.E. Brewington and G. Cybenko. Keeping up with the changing web. *Computer*, 33(5):52–58, 2000.

[7] D. Brickley and R.V. Guha. Resource description framework schema specification 1.0. Technical report, W3C, 2000.

[8] M.E. Califf and R.J. Mooney. Relational learning of pattern-match rules for information extraction. In *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, pages 6–11. AAAI Press, 1998.

[9] C.-K. Chang. Bidding against competitors. *IEEE Transactions on Software Engineering*, 16(1):100–104, 1990.

[10] S. Chawathe, H. García-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J.D. Ullman, and J. Widom. The TSIMMIS project: integration of heterogeneous information sources. In *Proceedings of the 16th Meeting of the Information Processing Society of Japan (IPSJ'94)*, pages 7–18, 1994.

[11] K. Ciesielski. *Set theory for the working mathematician.* Cambridge University Press, 1997.

[12] G. Cong, L. Yi, B. Liu, and K. Wang. Discovering frequent substructures from hierarchical semi-structured data. In *Proceedings of the 2nd SIAM International Conference on Data Mining (SDM'02)*, 2002.

[13] O. Corcho and A. Gómez-Pérez. Evaluating knowledge representation and reasoning capabilities of ontology specification languages. In *Proceedings of the ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods*, pages 1–9, 2000.

[14] O. Corcho and A. Gómez-Pérez. A road map on ontology specification languages. In *Proceedings of the ECAI'00 Workshop on Applications of Ontologies and Problem Solving Methods*, 2000.

[15] M. Craven, D. DiPasquo, D. Freitag, A.K. McCallum, T.M. Mitchell, K. Nigam, and Seán Slattery. Learning to construct knowledge bases from the world wide web. *Artificial Intelligence*, 118(1/2):69–113, 2000.

[16] H. Davulcu, S. Vadrevu, S. Nagarajan, and I.V. Ramakrishnan. OntoMiner: bootstrapping and populating ontologies from domain-specific web sites. *IEEE Intelligent Systems*, 18(1):24–33, 2003.

[17] M. Dean and G. Schreiber. OWL web ontology language reference, 2004.

[18] A. Diller. *Z: an introduction to formal methods.* John Wiley & Sons, 1994.

[19] L. Eikvil. Information extraction from world wide web - a survey. Technical Report 945, Norwegian Computing Center, 1999.

[20] F. Esposito, G. Semeraro, N. Fanizzi, and S. Ferilli. Multistrategy theory revision: induction and abduction in INTHELEX. *Machine Learning*, 38 (1-2):133–156, 2000.

[21] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua server: a tool for collaborative ontology construction. *International Journal of Human Computer Studies*, 46(6):707–727, 1997.

[22] D. Faure and C. Nédellec. A corpus-based conceptual clustering method for verb frames and ontology acquisition. In *Proceedings of the LREC workshop on Adapting Lexical and Corpus Resources to Sublanguages and Applications*, pages 5–12, 1998.

[23] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management (EKAW'00)*. Springer, 2000.

[24] D. Fensel, I. Horrocks, F. van Harmelen, D. L. McGuinness, and P. F. Patel-Schneider. OIL: an ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):293–310, 2001.

[25] D. Florescu, A.Y. Levy, and A. Mendelzon. Database techniques for the world wide web: a survery. *ACM SIGMOD Record*, 27(3):59–74, 1998.

[26] UCLA Center for Communication Policy. The UCLA internet report. Surveying the digital future: year three. Technical report, 2003.

[27] L. Francisco-Revilla, F. Shipman, R. Furuta, U. Karadkar, and A. Arora. Managing change on the web. In *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL'01)*, pages 67–76. IEEE Press, 2001.

[28] D. Freitag. Information extraction from HTML: application of a general machine learning approach. In *Proceedings of the 15th Conference on Artificial Intelligence (AAAI'98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98)*, pages 517–523. American Association for Artificial Intelligence, 1998.

[29] L.M. Fuld. *The new competitor intelligence: the complete resource for finding, analyzing, and using information about Your competitors*. John Wiley & Sons, 1994.

[30] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. DTD inference from XML documents: the XTRACT approach. *IEEE Data Engineering Bulletin*, 26(3):18–24, 2003.

[31] M.L. Ginsberg. Knowledge interchange format: the KIF of death. *Artificial Intelligence*, 12(3):57–63, 1991.

[32] A. Gómez-Pérez and O. Corcho. Ontology specification languages for the semantic web. *IEEE Intelligent Systems*, 17(1):54–60, 2002.

[33] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human Computer Studies*, 43 (5/6):907–928, 1993.

[34] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[35] N. Guarino. Understanding, building and using ontologies. *International Journal of Human-Computer Studies*, 46(2/3):293–310, 1997.

[36] H. Han and R. Elmasri. Ontology extraction and conceptual modeling for web information. In *Information Modeling for Internet Applications*, pages 174–188. Idea Group Publishing, 2003.

[37] S. Handschuh, S. Staab, and F. Ciravegna. S-CREAM - Semi-automatic CREAtion of Metadata. In *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management (CIKM'02)*, pages 358–372. Springer, 2002.

[38] S. Handschuh, S. Staab, and A. Maedche. CREAM: creating relational metadata with a component-based, ontology-driven annotation framework. In *Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001)*, pages 76–83. ACM Press, 2001.

[39] J. Heflin. *Towards the Semantic Web: Knowledge Representation in a Dynamic, Distributed Environment*. PhD thesis, University of Maryland, 2001.

[40] J. Heflin and J. Hendler. Dynamic ontologies on the web. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI'00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI'00)*, pages 443–449. AAAI Press, 2000.

[41] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2): 30–37, 2001.

[42] I. Horrocks. DAML+OIL: a reason-able web ontology language. In *Proceedings of the CAiSE 2002 workshop on Web Services, E-Business, and the Semantic Web (WES'02)*, page 174. Springer, 2002.

[43] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

[44] C.-N. Hsu. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In *Proceedings of the AAAI Workshop on AI and Information Integration*, pages 66–73, 1998.

[45] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.

[46] G. Huck, P. Fankhauser, K. Aberer, and E.J. Neuhold. Jedi: extracting and synthesizing information from the web. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (IFCIS'98)*, pages 32–43, 1998.

[47] D.E. Hussey, P.V. Jenster, and P. Jenster. *Competitor intelligence: turning analysis into success*. John Wiley & Sons, 1999.

[48] J. Jacky. *The way of Z: practical programming with formal methods*. Cambridge University Press, 1996.

[49] C.F. Kalmbach and D.M. Palmer. *eCommerce and alliances: how eCommerce is affecting alliances in value chain businesses*. Accenture LLP, 2002.

[50] P.D. Karp, V.K. Chaudhri, and J. Thomere. XOL: an XML-based ontology exchange language. `http://www.ai.sri.com/~pkarp/xol`, 1999.

[51] B. Katz and J.J. Lin. START and beyond. In *Proceedings of 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI'02)*, 2002.

[52] R.E. Kent. Conceptual knowledge markup language: the central core. In *Proceedings of the 12th Workshop on Knowledge Acquisition, Modeling and Management*, 1999.

[53] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[54] N. Kushmerick. Wrapper verification. *World Wide Web Journal*, 3(2):79–94, 2000.

[55] N. Kushmerick, D.S. Weld, and R.B. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 729–737, 1997.

[56] D. Lehmann. Nonmonotonic logics and semantics. *Journal of Logic and Computation*, 11(2):229–256, 2001.

[57] K. Lerman, S.N. Minton, and C.A. Knoblock. Wrapper maintenance: a machine learning approach. *Journal of Artificial Intelligence Research*, 18 (2003):149–181, 2003.

[58] A.Y. Levy and M.-C. Rousset. Combining horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1-2):165–209, 1998.

[59] L. Lim, M. Wang, S. Padmanabhan, J.S. Vitter, and R. Agarwal. Characterizing web document change. In *Proceedings of the 2nd Conference on Web-Age Information Management (WAIM'01)*, pages 133–144. Springer, 2001.

[60] M. Luck and M. d'Inverno. Autonomy: A nice idea in theory. In *Proceedings of the 7th International Workshop on Agent Theories, Architectures and Languages (ATAL'01)*, pages 351–354. Springer, 2001.

[61] S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology-based web agents. In *Proceedings of the 1st International Conference on Autonomous Agents (Agents'97)*, pages 59–68. ACM Press, 1997.

[62] R.M. MacGregor. Inside the LOOM description classifier. *SIGART Buletin*, 2(3):88–92, 1991.

[63] A. Maedche and S. Staab. Semi-automatic engineering of ontologies from text. In *Proceedings of the 12th International Conference on Software and Knowledge Engineering (KSI'00)*, 2000.

[64] D.L. McGuinness, R. Fikes, J. Hendler, and L.A. Stein. DAML+OIL: an ontology language for the semantic web. *IEEE Intelligent Systems*, 17(1): 72–80, 2002.

[65] D.L. McGuinness, R. Fikes, L.A. Stein, and J.A. Hendler. DAML-ONT: an ontology language for the semantic web. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, pages 65–93. MIT Press, 2003.

[66] G. Mecca, P. Merialdo, and P. Atzeni. ARANEUS in the era of XML. *IEEE Data Engineering Bulletin*, 22(3):19–26, 1999.

[67] M. Minsky. *A framework for representing knowledge.* McGraw-Hill, 1975.

[68] E. Motta. *Reusable components for knowledge modelling: case studies in parametric design problem solving.* IOS Press, 1999.

[69] I. Muslea, S. Minton, and C.A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114, 2001.

[70] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semi-structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 295–306. ACM Press, 1998.

[71] H.S. Nwana. Software agents: an overview. *Knowledge Engineering Review*, 11(3):205–244, 1995.

[72] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of the 19th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 35–46. ACM Press, 2000.

[73] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[74] B. Potter, J. Sinclair, and D. Till. *Introduction to formal specification and Z.* Prentice Hall, 1996.

[75] J. Powell. Spinning the world wide web: an HTML primer. *Database*, 18 (1):54–59, 1995.

[76] T. Powell. *HTML & XHTML: the complete reference.* McGraw-Hill, 2003.

[77] S. Powers. *Practical RDF.* O'Reilly, 2003.

[78] M.R. Quillian. Word concepts: a theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12(5):410–430, 1967.

[79] J. Reynolds and R. Mofazali. *The complete e-commerce book: design, build and maintain a successful web-based business.* CMP Books, 2000.

[80] S. Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[81] J.F. Sowa. *Conceptual structures: information processing in mind and machine.* Addison-Wesley, 1984.

[82] E. Spertus. ParaSite: mining structural information on the web. In *Selected papers from the 6th International Conference on World Wide Web*, pages 1205–1215. Elsevier Science Publishers, 1997.

[83] B. Starr, M.S. Ackerman, and M. Pazzani. Do-I-Care: tell me what's changed on the web. In *Proceedings of the AAAI Spring Symposium on Machine Learning in Information Access Technical Papers*, 1996.

[84] R. Studer, V.R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data Knowledge Engineering*, 25(1-2):161–197, 1998.

[85] D. Tsichritzis. *Electronic commerce.* Centre Universitaire d'Informatique (University of Geneva), 1998.

[86] M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna. MnM: ontology driven tool for semantic markup. In *Proceedings of the ECAI 2002 Workshop on Semantic Authoring, Annotation & Knowledge Markup (SAAKM'02)*, 2002.

[87] M.J. Wooldridge and M.R. Jennings. Intelligent agents: theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[88] M.J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, pages 71–80. ACM Press, 2002.

[89] M.J. Zaki and C.C. Aggarwal. XRules: an effective structural classifier for XML data. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, pages 316–325. ACM Press, 2003.

JOSÉ LUIS ARJONA
ACERCANDO EL CONOCIMIENTO EN LA WEB A
LOS AGENTES SOFTWARE

4 MARZO 2005

El Presidente,