HyCom

Desarrollo de Aplicaciones Hypermedia Mediante un Lenguaje Específico al Dominio

Trabajo de Grado

Licenciatura en Informática

Facultad de Informática

Universidad Nacional de La Plata

Alumnos:
Daniel H. Marcos
Walter A. Risi

Directores:
Pablo E. Martínez López
Gabriel A. Baum

Noviembre 1999

Agradecimientos

Si bien el trabajo presentado en esta tesis es producto de nuestro esfuerzo, creemos absolutamente que la misma no habría sido posible sin la colaboración y el soporte de mucha gente. Existen muchas personas a las que queremos agradecer, y trataremos de no olvidar a nadie. Desde ya, nos disculpamos si olvidamos de mencionar a alguien.

En primer lugar agradecemos a nuestros directores, Fidel y Gabriel, quienes siempre guiaron nuestros pasos en la carrera y en nuestro trabajo de investigación. Nos sentimos plenamente agradecidos de que su consejo no se haya limitado a nuestra formación académica, sino que también haya abarcado el aspecto personal, contribuyendo a nuestra integridad como personas, investigadores y profesionales.

No podemos dejar de mencionar a todos nuestros compañeros del LIFIA, porque siempre contribuyeron a que el ambiente de trabajo sea muy ameno y cómodo. Queremos agradecer especialmente a Anabella y Guillermo, quienes nos acompañaron en gran parte de nuestra carrera, resultando siempre excelentes compañeros y amigos.

Agradecemos fundamentalmente al LIFIA como institución, por proveernos un lugar de trabajo muy ameno y por ayudarnos tanto en lo personal como lo académico y lo económico. Queremos mencionar especialmente a Gustavo y a Fernando por brindarnos su ayuda para orientar nuestra carrera.

Finalmente, no podemos dejar de expresar nuestra gratitud a los jurados de esta tesis, quienes colaboraron de gran manera a agilizar su evaluación. De la misma manera, queremos agradecer al personal de la recientemente formada Facultad de Informática por aliviar y acelerar los trámites de presentación de este trabajo. Creemos que su colaboración resulta ejemplar en el crecimiento de nuestra flamante facultad y será sin duda un gran aporte hacia una universidad menos burocrática y de mejor nivel académico.

Agradecimientos Personales de Daniel

Mis mayores agradecimientos son para mi familia, que me ha apoyado muchísimo desde los inicios de mi carrera. Valoro enormemente el esfuerzo que han realizado por mí, tanto en lo económico como en lo personal, que a pesar de la distancia siempre me han hecho sentir muy acompañado.

Por otro lado, agradezco especialmente a Walter con quien he compartido todos estos años de estudio dentro de la facultad, resultando ser un excelente compañero y amigo. Por otro lado, agradezco a la gente del LIFIA por haber contribuído a que el ambiente de trabajo haya sido muy cómodo y agradable.

Finalmente, hay un montón de amigos que me gustaría nombrar y que me han hecho sentir a gusto durante estos años vividos en La Plata. Por ello agradezco entre otros a Anabella y Guillermo, Carolina, Fernando, Leandro, Claudia, Diana, Adriana, Mónica, Chule, Stella y Andrea.

Agradecimientos Personales de Walter

En primer lugar quiero expresar mi enorme gratitud a mi familia, que resultó la motivación constante para seguir siempre adelante y me guió en mi formación tanto profesional como humana. Por otro lado, quiero agradecer especialmente a Eliana, por su constante apoyo y por estar siempre a mi lado tanto en los mejores momentos como en los más difíciles.

No puedo dejar de expresar mi agradecimiento a Daniel, quien compartió conmigo todos estos años de estudio y trabajo. Me siento también muy agradecido con todos mis compañeros de la facultad y del LIFIA, especialmente con Anabella y Guillermo.

Finalmente, pero no en último lugar, quiero mencionar a mis amigos de siempre por haberme brindado su amistad y algunos de los mejores momentos de mi vida. Agradezco pues a Hernán L., George, Sonny, Miguel, Juan Ignacio E., Gustavo P. y Germán G. A todos ellos, muchas gracias.

Daniel H. Marcos
Walter A. Risi
La Plata, 10 de noviembre de 1999

Prefacio

La ingeniería de aplicaciones hypermedia es un campo que ha generado creciente interés en los últimos años. Existen múltiples razones que motivan este interés, entre las que se pueden mencionar el uso de tecnología multimedia en las aplicaciones y la incremental expansión de la *World Wide Web*.

Este trabajo enfoca el problema de la ingeniería de aplicaciones hypermedia mediante un lenguaje de dominio específico, embebido en un lenguaje funcional. Los lenguajes de dominio específico se han reconocido como una alternativa muy interesante para la ingeniería de aplicaciones a un alto nivel de abstracción. Por otro lado, los grandes avances en los lenguajes funcionales llevaron a que una de las mejores formas de implementar un lenguaje de dominio específico sea embebiéndolo en un lenguaje funcional.

La motivación del trabajo se basa en dos puntos fundamentales. Por un lado, aprovechar la metodología de los lenguajes embebidos en lenguajes funcionales para el desarrollo de hypermedia. Por otro lado, extender el campo de aplicación de los lenguajes funcionales a las aplicaciones con características hypermediales. De esta forma, esta tesis propone un intercambio de experiencia entre ambas áreas, resultando en beneficios concretos para las mismas.

Índice General

Capítulo 1

Introducción

El campo de las aplicaciones hypermediales ha crecido exponencialmente en los últimos años, principalmente a causa de la explosión de la *World Wide Web*. El crecimiento ha venido acompañado de diferentes enfoques para enfrentar el desarrollo de este tipo de aplicaciones. Existen varias alternativas para el desarrollo, desde la implementación directa hasta la utilización de modelos y metodologías de ingeniería.

Este trabajo de tesis presenta un enfoque al desarrollo de hypermedia mediante un lenguaje específico al dominio de hypermedia. En este capítulo reseñamos brevemente los conceptos involucrados en esta tesis y presentamos nuestra propuesta. Comentamos además las contribuciones de nuestro trabajo y la estructura general de la tesis.

1.1 El Problema del Desarrollo de Hypermedia

El desarrollo de hypermedia es un área que ha tenido mucho crecimiento en los últimos años, motivado principalmente por el rápido crecimiento de la *World Wide Web* (WWW) y sus implicaciones comerciales. El interés por el campo ha generado diversos enfoques para el desarrollo de aplicaciones hypermediales [SR95, ISB95, BN96, FNN96].

La comunidad académica, por un lado, identificó claramente los objetivos del desarrollo y propuso soluciones. De esta forma surgieron diferentes metodologías y modelos soportando los diferentes puntos. También se identificaron patrones de diseño y se los comenzó a utilizar en el desarrollo [RSG97].

La comunidad comercial enfrentó la necesidad por medio de todo tipo de herramientas, basadas principalmente en la implementación de los sistemas. Las herramientas van desde ambientes para la edición de páginas WWW (como FrontPage [Fro99], etc.) hasta soporte para aplicaciones *server-side* (como ColdFusion [Col99], WebObjects [Web99], etc.).

En la mayoría de los casos, los frentes comercial y académico avanzaron por sendas separadas. Los avances en la ingeniería de aplicaciones no fueron tenidos en

cuenta en gran parte por los productos comerciales, principalmente por su orientación al público general. De esta forma, los avances en la ingeniería de aplicaciones no son utilizados en todo su potencial, por la falta de soporte apropiado en los ambientes de implementación.

1.2 Lenguajes Específicos al Dominio

Los lenguajes específicos al dominio (*Domain Specific Languages*, o DSLs) han existido por varios años [Hud96a]. Este tipo de lenguajes, a diferencia de los lenguajes de propósito general, capturan exactamente la semántica del dominio de aplicación elegido. Esta aparente restricción permite ganar expresividad en el lenguaje, cuyas construcciones están elegidas exclusivamente con un dominio de aplicación en mente.

Recientemente, la ingeniería de *software* por medio de DSLs ha recibido gran atención; de hecho, han surgido conferencias dedicadas exclusivamente a este enfoque del desarrollo de *software*. Uno de los enfoques más difundidos consiste en el de DSL embebido (*Domain Specific Embedded Language*, o DSEL), que se basa en embeber un vocabulario específico en un lenguaje existente (normalmente de propósito general).

El enfoque de DSEL presenta interesantes ventajas. Por un lado, se evita tener que definir un nuevo lenguaje con cada dominio nuevo a tratar. Además, los lenguajes pueden aprovechar la sintaxis y semántica del lenguaje base (o lenguaje huésped). Como resultado, se tiene una curva de aprendizaje reducida, ya que los usuarios no tienen que familiarizarse con un lenguaje completamente diferente para tratar con diferentes DSELs. Por otro lado, permite el reuso no sólo de las decisiones de diseño del lenguaje base, sino que permite la integración directa con otros DSELs preexistentes embebidos en el mismo.

La elección de un lenguaje base no es trivial y debe realizarse cuidadosamente. En general, se prefiere lenguajes con gran capacidad de extensibilidad y gran expresividad. En el primer caso se desea que el lenguaje permita incorporar nuevas construcciones al mismo nivel que las construcciones primitivas. En el segundo caso, se desea que el lenguaje sea suficientemente expresivo para poder especificar las construcciones del dominio con claridad, sobre todo porque los lenguajes estarán orientados a expertos en el dominio y no necesariamente expertos programadores.

La experiencia ha demostrado que el lenguaje Haskell [PJH99] es una excelente opción a la hora de embeber DSELs [LM99]. Diversos experimentos que van desde aplicaciones militares [CHJ93] hasta gráficos [EH97] y música [Hud96b] han dado resultados muy satisfactorios.

1.3 HyCom: Un DSEL para Hypermedia

El trabajo que presentamos en esta tesis se denomina HyCom (por **Hy***permedia* **Com***binators*), y consiste en la definición e implementación de un DSEL embebido

en Haskell para el desarrollo de hypermedia.

Diseñar un DSEL para hypermedia nos permite tener en cuenta desde un principio las necesidades fundamentales de este campo. Consideraciones importantes como el soporte para los enfoques de ingeniería y la prototipación temprana pueden soportarse directamente. Además, el DSEL actúa como intermediario entre los modelos de alto nivel y la implementación, reduciendo la habitual distancia entre estos dos mundos.

El lenguaje elegido como base es Haskell [PJH99], debido a las características atractivas que ofrece. Haskell es un lenguaje declarativo, puramente funcional y posee un gran número de características interesantes como alto orden, clases para polimorfismo paramétrico, sistema de módulos y compilación eficiente [GHC99]. Más aún, existe amplia experiencia en el desarrollo de DSELs con Haskell [Hud96a]. Esto último nos provee además con una gran cantidad de DSELs existentes, los cuales pueden ser integrados inmediatamente con HyCom.

HyCom toma como principio fundamental de diseño la *programación por combinación*. En este enfoque una aplicación se construye a partir de componentes simples, transformadores y combinadores. Un componente simple puede ser por ejemplo un *widget*. Un transformador es una función que agrega o modifica cierta funcionalidad del componente (por ejemplo, agregar barras de desplazamiento al *widget*). Finalmente, un combinador permite tomar dos o más componentes y combinarlos para formar un componente más complejo (por ejemplo, combinar una imagen y un componente de texto para formar una imagen etiquetada). La programación por combinación es una metodología muy efectiva y probada en los lenguajes declarativos [CH98]. Los diseños resultantes con HyCom pueden ser compilados, resultando en aplicaciones en diferentes plataformas (como la WWW u otras).

1.4 Contribuciones del Trabajo

Las contribuciones más relevantes del trabajo pueden resumirse en los siguientes puntos.

- Aprovechar la metodología DSEL y la experiencia en el diseño de DSELs embebidos en lenguajes declarativos para el desarrollo de aplicaciones hypermedia. El lenguaje resultante se beneficia al ser diseñado con las necesidades de hypermedia como primer objetivo. Por otro lado, la experiencia existente en los DSELs embebidos permite reusar no sólo decisiones de diseño, sino también DSELs preexistentes.

- Realizar un modelo para el desarrollo de hypermedia en lenguajes declarativos. Si bien estos lenguajes han sido tradicionalmente relegados a aplicaciones puramente académicas, lenguajes modernos (como Haskell, Clean [Cle99] o ML [Aug84]) han llegado a un grado de madurez suficiente para empezar a ser aplicados a situaciones reales. El desarrollo de un modelo hypermedial

declarativo es un gran aporte para estos lenguajes, ya que la funcionalidad hypermedial en las aplicaciones comienza a ser un requisito fundamental (sobre todo en el desarrollo de aplicaciones para la WWW).

Las contribuciones de la tesis son, por lo tanto, en dos sentidos. El desarrollo de hypermedia se beneficia de la utilización de un DSEL declarativo. El lenguaje base se beneficia al enriquecer su repertorio de DSELs, incorporando la capacidad de desarrollar aplicaciones hypermediales en él.

1.5 Evolución de HyCom

El desarrollo de HyCom pasó por varias etapas desde su concepción inicial. La idea de un DSEL para hypermedia surge a partir de observar los interesantes resultados en la utilización de una metodología composicional en el desarrollo de aplicaciones [CH98, Hud96b, FPJ96, HM96].

Durante el diseño de la primera versión de HyCom focalizamos nuestro trabajo en expresar aspectos de aplicaciones hypermediales en Haskell. Esta etapa del trabajo tuvo como motivación principal aprovechar la experiencia existente en el uso de combinadores en el diseño de DSELs y analizar su viabilidad de aplicación al dominio hypermedial. Los resultados de esta primera fase del trabajo se resumen en un *paper* presentado en la *Segunda Conferencia Latinoamericana de Programación Funcional (CLaPF97)* [MMLR97].

La siguiente etapa de nuestro trabajo consistió en el análisis de los aspectos metodológicos de HyCom en el desarrollo de aplicaciones (es decir, qué pasos deben seguirse al definir una hypermedia mediante el DSEL). En esta fase también nos ocupamos de desarrollar aplicaciones reales mediante el DSEL (ver Apéndice C), a partir de lo cual enriquecimos nuestra experiencia en el dominio. Describimos los resultados de esta etapa del trabajo en un *poster* presentado en *III International Conference on Functional Programming (ICFP98)* [MMLR98].

Aprovechando el *feedback* obtenido en las presentaciones en conferencias y la experiencia ganada con el uso real de HyCom, nos planteamos una tercera etapa en el desarrollo. En esta fase decidimos realizar un completo rediseño del DSEL, para lograr mayor flexibilidad y a la vez riqueza conceptual. Resultados preliminares de esta fase fueron mostrados en el *V Congreso Argentino de Ciencias de la Computación (CACIC99)* [RMML99].

Esta tesis presenta los resultados generales de nuestro trabajo en HyCom, focalizándose particularmente en la versión del DSEL definida en la última etapa. Las etapas anteriores están presentadas en mayor o menor medida en la etapa final.

1.6 Esquema General de la Tesis

El presente trabajo se encuentra dividido en diferentes partes. Cada parte cubre un aspecto del trabajo; el orden de lectura de las mismas no debe hacerse necesariamente en forma secuencial. Aclaramos que esta tesis tiene un fuerte basamento en

el paradigma funcional y el lenguaje Haskell, asumiéndose un conocimiento básico de los mismos. Recomendamos al lector que necesite reforzar su conocimiento de estos temas que consulte el Apéndice A o la bibliografía antes de proseguir con la lectura.

La Parte I cubre los conceptos preliminares necesarios para el entendimiento general del trabajo. El Cap. 2 resume los aspectos más importantes relativos a hypermedia y los enfoques existentes para su desarrollo. Por su parte, el Cap. 3 introduce los puntos relevantes sobre la metodología DSL, así como sus ventajas y consideraciones especiales a la hora de diseñar un DSL. La lectura de esta parte puede ser omitida o realizarse parcialmente, en base al conocimiento previo de los lectores en estas áreas.

La contribución principal de la tesis se presenta en la Parte II. El Cap. 4 presenta el DSEL HyCom, los principios generales de su diseño, su estructura general y los diferentes tipos de componentes que lo constituyen. Por su lado, el Cap. 5 introduce el soporte a plataformas específicas de implementación y las funciones de compilación. El lector que no tenga conocimiento previo de los conceptos generales de HyCom debería comenzar su lectura de la tesis por esta parte. La lectura de esta parte requiere un conocimiento básico de los conceptos de hypermedia y DSL, los cuales se resumen en la parte anterior. El lector que tiene un conocimiento general de HyCom y prefiere centrarse en su utilización, puede omitir la lectura de esta parte y referirse a la parte siguiente.

La utilización de HyCom en el desarrollo de aplicaciones es el tema de la Parte III. En primer lugar, el Cap. 6 muestra la forma en que una aplicación se desarrolla mediante HyCom, esbozando un ciclo de desarrollo de aplicaciones y un ejemplo sencillo. En el Cap. 7 presentamos el concepto de biblioteca de autoría, que resulta de la utilización de la metodología DSEL en el desarrollo de aplicaciones. A continuación, en el Cap. 8 mostramos un ejemplo real de la utilización de HyCom y las bibliotecas de autoría en el desarrollo de aplicaciones. Por su parte, el Cap. 9 muestra un enfoque al uso de HyCom en conjunción con una conocida metodología de diseño de hypermedia. El lector interesado en la forma en que HyCom se utiliza en el desarrollo de aplicaciones puede focalizarse en esta parte de la tesis. Es necesario aclarar que un conocimiento general de HyCom es requisito para la lectura de esta parte.

Finalmente, la Parte IV resume los comentarios finales resultantes de la realización de este trabajo. Comenzamos en el Cap. 10 haciendo una revisión de los trabajos relacionados con HyCom y comparándolos con él. A continuación, el Cap. 11 detalla las líneas de trabajo que surgen de la realización de esta tesis. Para finalizar, en el Cap. 12 resumimos las conclusiones obtenidas como resultado de la realización de este trabajo.

Como complemento de la tesis, la Parte V presenta tres apéndices. El Apéndice A resume en forma breve los conceptos básicos del paradigma funcional y el lenguaje Haskell. El Apéndice B, por su parte, provee un índice de las características de HyCom, sirviendo de referencia para el lector interesado en su utilización o en aspectos técnicos. Finalmente, el Apéndice C muestra algunos usos reales de

HyCom.

Parte I

Conceptos Preliminares

Capítulo 2

Aplicaciones Hypermediales

2.1 Introducción

El término *hypermedia* [Nie95] describe un conjunto de tecnologías destinadas a organizar la información, proveer asociaciones entre piezas relacionadas de información y visualizarlas en forma adecuada. En los últimos años, el interés en utilizar la tecnología de hypermedia en sistemas de información ha crecido exponencialmente. Consecuentemente, las aplicaciones hypermedia se han ido tornando más y más complejas.

El aumento de la complejidad de este tipo de aplicaciones ha motivado un fuerte interés en la creación de herramientas de desarrollo, tanto conceptuales como de *software*. Mientras que los desarrollos iniciales se hacían implementando directamente la aplicación, los sistemas actuales requieren enfoques más estructurados. Debido a esto, tanto las empresas comerciales como los investigadores de diversas áreas han desarrollado diversos productos, desde complejos ambientes de desarrollo hasta metodologías y modelos formales, que permiten enfocar el desarrollo de hypermedia como una tarea de ingeniería de *software*.

No obstante la amplia proliferación de herramientas de *software* y modelos para el desarrollo de hypermedia, el problema del desarrollo no ha sido completamente solucionado. Si bien los modelos y metodologías proveen abstracciones de alto nivel, el mapeo de las mismas a las herramientas de *software* existentes resulta en una pérdida de riqueza notable. Asimismo, los nuevos ambientes de desarrollo proveen nuevas facilidades, pero siguen sin enfocar el problema de una manera estructurada. Por otro lado, existen aspectos importantes que se relacionan con la integración de las herramientas y modelos, como la consideración especial de aspectos cognitivos y prototipación temprana. Estos problemas han hecho que el estudio de integración entre herramientas y modelos sea una importante área de investigación.

En este capítulo trataremos los problemas mencionados. Comenzaremos presentando el concepto de hypermedia, y viendo algunos de los sistemas más difundidos. Veremos luego cuáles son los puntos importantes para su desarrollo. Haremos lue-

go un resumen de los distintos enfoques existentes. Finalmente, analizaremos los problemas que aún persisten y las posibles soluciones.

2.2 Hypermedia

Hypermedia es una tecnología utilizada para organizar y proveer acceso a información de naturaleza multimedial. En vez de organizarse la información en forma secuencial, como en un libro, la información es agrupada en unidades denominadas *nodos*. Los nodos que están relacionados entre sí mediante entidades conceptuales llamadas *links*. En la figura Fig. 2.1 mostramos un esquema de un sistema de hypermedia.

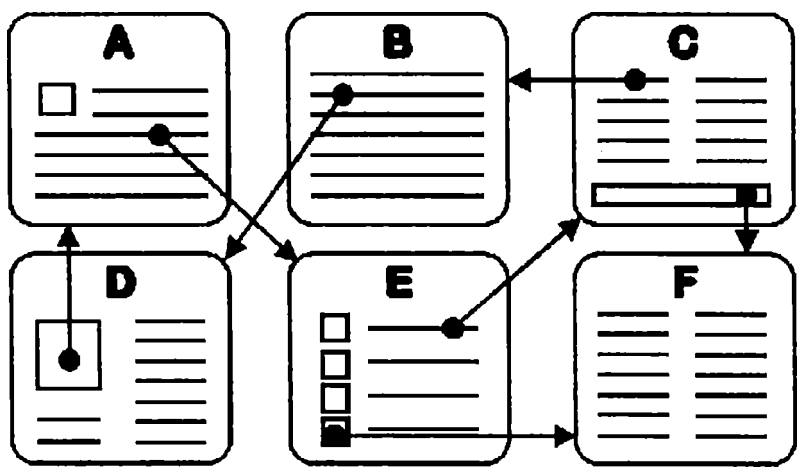

Figura 2.1: Estructura de un Sistema Hypermedial

El usuario obtiene la información accediendo a los nodos, y siguiendo los *links* según sus intereses, en vez de seguir un flujo secuencial. Contrástese esto con los medios tradicionales, como los impresos, en los cuales el usuario sigue una secuencia fija de páginas, o debe remitirse a un índice. Claramente, esta tecnología provee grandes ventajas a la hora de acceder a grandes flujos de información.

Las ventajas provistas por estas tecnologías han generado un creciente interés en los últimos años. Diferentes aplicaciones se han mostrado como muy adecuadas para un tratamiento hypermedial, como por ejemplo, libros electrónicos, tutoriales interactivos, catálogos, etc. La *World Wide Web* (WWW) se muestra hoy en día como el ámbito más importante para la proliferación de aplicaciones hypermediales de gran escala.

2.3 Puntos Importantes en el Desarrollo

El crecimiento del interés en las aplicaciones hypermedia ha generado un amplio mercado para las mismas. Las aplicaciones hypermedia crecen continuamente en complejidad y se hacen cada vez más necesarios los enfoques adecuados para su desarrollo.

Podemos distinguir varios puntos importantes a tener en cuenta en la autoría de este tipo de sistemas. Algunos de ellos tienen su par en el desarrollo de *software* tradicional, pero otros son exclusivos de las aplicaciones hypermedia. Veremos a continuación los puntos más relevantes.

2.3.1 Modelos y Métodos

Se necesitan modelos bien definidos para el desarrollo de hypermedia que permitan especificar las propiedades del sistema sin ambigüedades. Un modelo de este tipo nos permitiría especificar la estructura de la hypermedia en un alto nivel de abstracción, sin remitirnos directamente a la implementación. El modelo debe tener además un alto nivel de expresividad, de forma que el autor pueda definir adecuadamente el diseño de la aplicación.

También se requiere un enfoque estructurado en el desarrollo, mediante el cual puedan enfocarse las diferentes etapas del mismo en forma organizada. El desarrollador debería poder enfocar los distintos aspectos de la hypermedia, como la estructura navegacional o la presentación, en forma independiente. Deberían tenerse además guías generales que indiquen cómo ir desarrollando los diferentes aspectos, y cómo se relacionan entre sí.

La necesidad de métodos y modelos se ve reflejada en el desarrollo de todo tipo de *software*. La ingeniería de *software* se ha ocupado de esto durante años, de lo cual han surgido métodos para el desarrollo de software. Existen diversos tipos de metodologías y modelos, cubriendo desarrollos de *software* de diversos tipos, por ejemplo: Análisis y Diseño Estructurado [You91, GS79], Análisis y Diseño Orientado a Objetos [Boo91, Jac94, R+91], etc. Las metodologías tradicionales, sin embargo, se muestran insuficientes a la hora de utilizarlas para desarrollar aplicaciones hypermedia. Existen aspectos únicos de estas aplicaciones, como por ejemplo las consideraciones de navegación, que deben afrontarse en forma especial.

2.3.2 Validación y Evaluación

La validación y evaluación de especificaciones son deseables en todo desarrollo y no siempre son tenidas en cuenta. Existen diversas formas de evaluar y medir la correctitud y calidad de un diseño de *software*. Estas van desde las puramente formales, hasta las estadísticas y heurísticas. El dominio de aplicaciones hypermedia tiene sus propias necesidades en este sentido, como por ejemplo, evaluar la estructura navegacional [TM98].

2.3.3 Herramientas de Software

En general, los proyectos de desarrollo tienen que hacerse bajo presiones de tiempo. Es necesario contar con herramientas que permitan realizar las aplicaciones en forma efectiva y eficiente.

Este punto es común también al desarrollo de todo tipo de *software*. Muchas veces los sistemas deben completarse bajo estrictas restricciones de tiempo, teniéndose a veces que sacrificarse el tiempo necesario para un correcto diseño. Las metodologías y modelos deben integrarse de forma adecuada a las herramientas de desarrollo existentes.

2.3.4 Consideraciones Especiales

Finalmente, existen puntos que son exclusivos de los sistemas de hypermedia. Las aplicaciones de este tipo tienen altos requerimientos de usabilidad. Los factores humanos juegan un rol importante, y los mismos deben tenerse en cuenta a lo largo del proceso de diseño. Es por eso que se requieren facilidades para elaborar prototipos incrementalmente, realizándose testeos de la aplicación desde las primeras etapas del desarrollo. Esta consideración por factores humanos va más allá de los métodos de ingeniería, y requiere de ambientes especiales que faciliten la prototipación y testeo [NN95].

2.4 Enfoques Existentes para el Desarrollo

Las necesidades destacadas en la sección anterior han motivado el surgimiento de distintos tipos de herramientas para el desarrollo. Los enfoques varían, pasando de los eminentemente prácticos hasta los enfoques profesionales de ingeniería. A continuación veremos, en forma resumida, las principales características de las alternativas existentes.

2.4.1 Implementación Directa

La implementación directa constituye el enfoque más primitivo y quizás por eso es el más utilizado. Los usuarios realizan la implementación de la aplicación directamente utilizando herramientas WYSIWYG (*What You See Is What You Get*, donde el énfasis está puesto en los aspectos visuales), y no se requieren pasos previos de análisis y diseño cuidadosos. Las razones de la proliferación de este tipo de desarrollo son varias. Por un lado, la necesidad de obtener resultados rápidamente muchas veces desalienta al desarrollador a utilizar enfoques más elaborados. Por otro lado, muchas aplicaciones hypermedia (sobre todo en la WWW) son hechas por personas sin formación ingenieril, que desconocen prácticamente la existencia de métodos más avanzados. Otro factor importante es que los métodos ingenieriles sufren de una gran falta de publicidad, mientras que los ambientes WYSIWYG (como FrontPage [Fro99]) son ampliamente publicitados en diversos medios.

El enfoque de implementación directa es decididamente muy precario y es inadecuado para aplicaciones grandes. Para ello han surgido entornos que facilitan la implementación de aplicaciones en gran escala. Para el caso particular de la WWW, ambientes orientados al desarrollo de *sites* (como NetObjects [Net99] Fusion, etc.), permiten mantener un *look-and-feel* uniforme de las páginas. El uso de estas herramientas no representa realmente una evolución con respecto al enfoque básico, sino una disminución en la dificultad de la tarea de implementación. El énfasis sigue estando en la apariencia y no en la estructura.

2.4.2 Enfoques de Ingeniería

La necesidad de herramientas de alto nivel y la insuficiencia de los métodos tradicionales de ingeniería han motivado la aparición de diversas metodologías y modelos para el desarrollo de hypermedia. Estos métodos de ingeniería prescriben modelos bien definidos de los diferentes elementos de una aplicación. A su vez, describen el proceso de desarrollo como una serie de etapas, cada una enfocando un aspecto particular del sistema.

El método OOHDM (*Object Oriented Hypermedia Design Method*) [SR95] se basa en el paradigma de orientación a objetos. El mismo encara el desarrollo como una serie de cuatro pasos: modelado conceptual, diseño navegacional, diseño de interfaz abstracto e implementación. Estos pasos se realizan en una mezcla de estilos (iterativo, incremental y por prototipos). La metodología se describe en detalle en [Ros95].

Por otro lado, el método RMM (*Relationship Management Methodology*) [ISB95] se basa en el modelo de entidades y relaciones (ER) [EN90], muy utilizado para realizar modelos conceptuales de bases de datos. La metodología prescribe la utilización de un diagrama ER para capturar el dominio conceptual de la aplicación; luego se utilizan primitivas propias del método para derivar la estructura navegacional a partir del modelo conceptual (RMM indica un total de siete fases para el desarrollo de la aplicación pero sus autores han definido formalmente sólo aquellas referentes al modelo conceptual y el navegacional). La versión más actualizada de RMM se describe en [IKK98].

Existen otras metodologías y modelos, que encaran el desarrollo desde diferentes perspectivas [PV95, BN96]. En la comunidad de hypermedia no existe un consenso global sobre cuál método debe usarse en cada caso, pero en general se acepta que cada método se adecua mejor para un tipo de aplicación particular.

Como comentario final, destacamos que existen varios proyectos de herramientas CASE [Mar82] para el desarrollo de hypermedia. El ambiente RMCASE [DI95] soporta el desarrollo mediante RMM, y OOHDM-Case [LR96] respectivamente con OOHDM. Existen algunos otros proyectos de herramientas CASE que no llegaron aún a una fase de concreción importante [Ash96, BN96].

2.4.3 Consideraciones por los Factores Humanos

Las consideraciones por los factores humanos mencionados han influenciado el desarrollo de herramientas apropiadas. Los puntos más frecuentemente atendidos son la generación eficiente de prototipos (que permitan la evaluación temprana del diseño) y la posibilidad de alternar entre enfoques *top-down* y *bottom-up* en el desarrollo. El trabajo realizado por Marc Nanard y Jocelyne Nanard es una de las contribuciones más importantes al reconocimiento de los factores humanos en el desarrollo [NN95].

El ambiente *PageJockey* [FNN96] fue realizado con esta consideración especial como motivación principal. El sistema está pensado para realizar un desarrollo mediante prototipos incrementales, sin seguir una metodología estricta.

Las herramientas CASE que ya mencionamos también fueron realizados considerando estos aspectos. Por ejemplo, RMCase permite generar un prototipo en HTML.

2.5 Resumen

A pesar de la amplia proliferación de nuevas herramientas para el desarrollo de hypermedia, como métodos y ambientes de *software*, existen aún problemas que no han sido del todo solucionados. Estos problemas son motivo de investigación por la comunidad científica y profesional.

Por un lado, los nuevos ambientes de *software* no resuelven el problema del desarrollo. Estas herramientas proveen importantes facilidades, pero las mismas sólo facilitan la tarea, sin elevar el nivel del método de desarrollo. Por ejemplo, las herramientas orientadas al *site* de la WWW facilitan el mantenimiento pero no elevan el nivel de abstracción del desarrollo.

Los métodos y modelos de ingeniería son un enfoque muy profesional al desarrollo. Sin embargo, el traslado de los modelos a las herramientas de implementación corrientes suele resultar en una notable pérdida de riqueza. Un ejemplo claro es la WWW, donde por ejemplo un modelo orientado a objetos (como sería uno desarrollado en OOHDM) se pierde completamente al volcarse a páginas.

En resumen, concluímos que no obstante la amplia variedad de alternativas para el desarrollo aún existen puntos débiles. Nuestra opinión al respecto es que estas falencias radican en la excesiva separación entre los métodos formales de desarrollo y las herramientas utilizadas para la implementación. Creemos que una solución factible consiste en el desarrollo de una herramienta de especificación formal que permita derivar aplicaciones directamente de la especificación; de esta forma, el usuario podría realizar el diseño en un alto nivel de abstracción, obteniendo una implementación funcional sin necesidad de recurrir a herramientas de bajo nivel.

Capítulo 3

Lenguajes Específicos al Dominio

Cuando se piensa en un lenguaje de programación generalmente se lo hace sobre aquellos que son de propósito general, es decir, los que permiten desarrollar todo tipo de aplicación con relativamente el mismo grado de expresividad y eficiencia. Existe una manera mucho más natural de expresar una solución a un problema de un determinado dominio y consiste en utilizar un lenguaje específico al dominio en cuestión. HyCom aprovecha este enfoque para proveer un lenguaje específico al dominio de hypermedia.

En este capítulo presentamos el enfoque de lenguajes específicos al dominio (DSLs) para el desarrollo de software. Comenzamos comentando las características básicas de un DSL. Explicamos cuál es la metodología a seguir en el desarrollo de aplicaciones mediante un DSL. Mostramos luego las ventajas conseguidas por el enfoque y las eventuales desventajas que pueden surgir. Finalmente, analizamos las ventajas que pueden obtenerse al utilizar un enfoque de lenguaje embebido en la definición de un DSL.

3.1 Conceptos Básicos

Uno de los preceptos más afianzados en toda la comunidad de ingeniería de *software* es el hecho de que la *abstracción* es un factor de suma importancia en el desarrollo de programas. La evolución de los lenguajes de programación ha ido acompañada por un creciente soporte por los mecanismos de abstracción; entre los mecanismos más difundidos se encuentran los procedimientos, las funciones, los tipos de datos abstractos, los objetos, etc.

Los mecanismos de abstracción mencionados son en principio de propósito general, es decir, no están pensados para un dominio de aplicación determinado. Desde hace tiempo, sin embargo, se ha empezado a reconocer que una mejor forma de representar un dominio consiste en utilizar un lenguaje específico al mismo; en un lenguaje de este tipo, se proveen abstracciones que capturan pura y exclusivamente el dominio conceptual de la aplicación. Creemos oportuno citar las palabras del prestigioso investigador Paul Hudak [Hud96a].

> A pesar de que la generalidad es buena, podemos preguntarnos cuál es la abstracción "ideal" para una aplicación particular. En mi opinión, la abstracción ideal es un lenguaje de programación que está diseñado precisamente para esa aplicación: uno en el cual una persona puede desarrollar un sistema de software completo en forma rápida y efectiva. No es en absoluto general; debería capturar precisamente la semántica del dominio de aplicación, nada más ni nada menos. En mi opinión, un lenguaje específico al dominio es la "abstracción definitiva".

La idea de utilizar DSLs en el desarrollo de software no es en absoluto nueva y de hecho se encuentra muy difundida. La mayoría de los personas que están en contacto con la informática (no necesariamente programadores) ha utilizado un DSL de un algún tipo. Por ejemplo, el lenguaje HTML [HTM99] es un DSL ampliamente utilizado para el desarrollo de páginas en la *World Wide Web*. Otro DSL muy difundido es SQL, utilizado para efectuar consultas sobre bases de datos relacionales; Tcl y Tk son muy usados en entornos UNIX para el *scripting* del interfaces del usuario gráficas; las diferentes variantes de IDL [COM99, COR99] son DSLs para la descripción de componentes de *software*. Finalmente, y a modo de ejemplo de la ubicuidad de los DSLs, esta tesis ha sido escrita en LaTeX, un DSL para la descripción de documentos.

3.2 Método de Desarrollo mediante un DSL

El desarrollo de software mediante un DSL abarca varias etapas, que se relacionan en mayor o menor grado con el desarrollo del DSL mismo. Si el DSL es definido por el mismo grupo de desarrolladores que lo utiliza, es normal que el grupo decida refinar la definición del lenguaje en base a la experiencia ganada a partir de su uso. Si el DSL ha sido desarrollado por terceros (por ejemplo, si es un producto comercial) no necesariamente el desarrollo de aplicaciones tiene influencia en la evolución del DSL mismo (ya que no todos los desarrolladores aportarán datos a los diseñadores del lenguaje).

Si consideramos el ciclo de desarrollo comienza con la definición del DSL mismo, podemos resumir los pasos del ciclo en los siguientes puntos.

1. Se realiza un análisis del dominio, restringiéndolo e identificando las abstracciones clave del mismo.

2. Se diseña un DSL que captura precisamente la semántica del dominio de aplicación.

3. Se construyen herramientas de soporte al DSL (compiladores, intérpretes, etc.).

4. Se desarrollan aplicaciones (instancias del dominio) mediante el DSL. Esta fase incluye las etapas convencionales de todo proyecto de software, como el

análisis, diseño, implementación, etc. Claramente, fases como la de análisis se facilitan drásticamente, puesto que ya se realizó gran parte del análisis del dominio durante el desarrollo del DSL.

Las primeras etapas, definición y restricción del dominio y diseño del DSL, son la clave para la aplicación exitosa de la metodología. Como todo ciclo de desarrollo de *software*, el método de desarrollo propuesto no se hace en forma completamente secuencial, sino que cada etapa está sujeta a revisiones que puede llevar a volver a una etapa previa del ciclo.

El ciclo que presentamos asume que el desarrollo comienza con la definición del DSL. Es claro que realizaremos estos primeros pasos sólo una vez, eventualmente volviendo a realizarlos en forma parcial para efectuar refinamientos sobre el DSL. Aclaramos también que muchos desarrolladores probablemente no tendrán contacto con los diseñadores del lenguaje; para ellos, el ciclo de desarrollo se simplificará a la última fase.

3.3 Ventajas y Desventajas de los DSLs

Cuando tomamos la decisión de usar un DSL para desarrollar una aplicación, obtenemos numerosas ventajas. En la mayoría de los casos, las ventajas son una consecuencia directa de la representación concisa y específica del dominio de aplicación. Existen también algunos peligros que debemos considerar al adoptar el enfoque DSL, para evitar que se conviertan en eventuales desventajas.

Un programa escrito mediante un DSL es mucho más conciso que uno escrito en un lenguaje de propósito general, debido a que sólo se utilizan abstracciones correspondientes al dominio de aplicación. El hecho de que los programas sean más concisos hace que los mismos sean más fáciles de comprender y consecuentemente de mantener. Además, como utilizan solamente elementos de un dominio en particular, pueden ser usados por expertos en ese dominio, que no necesariamente son programadores. A modo de ejemplo, el siguiente es un fragmento de código HTML especificando un *anchor*, que al activarse lleva al navegador al *site* del LIFIA.

```
<A HREF="www.lifia.unlp.edu.ar"> Ir al Site del LIFIA </A>
```

El código anterior especifica una porción de una página WWW que permite navegar a un *site* posiblemente remoto. La navegación implicará probablemente establecer una conexión con un *server* remoto, iniciar una transacción mediante el protocolo HTTP, etc. Sin embargo, el usuario puede ignorar estos detalles al utilizar el DSL; utilizando el mismo especifica el comportamiento de la aplicación al nivel de abstracción adecuado y los aspectos de bajo nivel son manejados por el software de soporte al DSL (en este caso, el *web browser* y el *web server*).

En general, los programas hechos con DSLs llevan menor tiempo para desarrollarse. Esto se debe a que, al ser específicos a un dominio, muchas tareas propias del mismo son manejadas automáticamente por el lenguaje. Por ejemplo, en SQL

hacemos una consulta especificando qué es lo que queremos recuperar, sin preocuparnos en la estructura de archivos en la que se almacena la base de datos, si está distribuída o no, etc. El siguiente código SQL recupera todos los clientes que viven en Buenos Aires de una base de datos.

```
SELECT * FROM CLIENTES
WHERE CIUDAD = "BUENOS AIRES"
```

Realizar una consulta similar utilizando un lenguaje de propósito general requiere en primer lugar establecer una conexión con la base de datos, realizar la iteración sobre los elementos de la tabla, chequear la condición en cada paso de la iteración y eventualmente cerrar la base de datos. En el caso de que tuviéramos consideraciones adicionales, como la distribución de la base de datos, el código se complica aún más. Usar un DSL como SQL nos permite ignorar gran parte de esos detalles, que en general son resueltos por el manejador de bases de datos.

Otra de las características importantes que hace poderosos a los DSLs es la posibilidad de desarrollar herramientas al dominio al cual se aplican. Por ejemplo, para el lenguaje LaTeX existe una herramienta denominada BibTeX que provee una forma estructurada para mantener las referencias bibliográficas. Otro ejemplo lo constituyen los optimizadores de consultas para SQL.

A pesar de sus numerosas ventajas, el enfoque DSL también esconde algunos peligros potenciales que debemos tener en cuenta. En primer lugar, si desarrollamos un lenguaje completamente nuevo por cada dominio de aplicación, corremos el riesgo de tener una *Torre de Babel*: cada vez que queremos desarrollar un programa para un dominio nuevo, tenemos que aprender un lenguaje diferente. Por otro lado, diseñar completamente un lenguaje es una tarea de gran envergadura y dista mucho de ser trivial; el enfoque DSL corre el riesgo de implicar costos de desarrollo iniciales inviablemente altos.

3.4 Lenguajes Embebidos Específicos al Dominio

Los mencionados problemas de la metodología DSL fueron bien identificados y estudiados por la comunidad de desarrolladores. La solución adoptada es en principio muy simple: debido a que no deseamos construír un lenguaje nuevo por cada dominio, es más conveniente heredar la infraestructura de un lenguaje existente (adaptándolo adecuadamente al dominio de interés) resultando en el enfoque de DSL *embebido* (DSEL).

3.4.1 El Enfoque DSEL

El enfoque DSEL consiste en embeber un DSL en un lenguaje de propósito general ya existente, al que llamamos *lenguaje base*. El hecho de reutilizar la arquitectura de un lenguaje nos permite heredar su estructura semántica y sintáctica, además de reutilizar todas las decisiones de diseño tomadas en su desarrollo. Por otro

lado, muchas de las herramientas existentes para el lenguaje base (compiladores, intérpretes, etc.) pueden reutilizarse directamente. El primer problema de la metodología DSL (los costos de desarrollo iniciales) se soluciona aceptablemente de esta forma.

Utilizar un DSEL también implica poder aprovechar la experiencia previa en el lenguaje base; un programador experto en el lenguaje base verá disminuida de gran manera la curva de aprendizaje de DSELs embebidos en el mismo. De forma similar, una vez que se conoce un DSEL embebido en cierto lenguaje, el aprendizaje de otros DSELs embebidos en el mismo lenguaje también se facilita. La segunda desventaja de la metodología DSL (la curva de aprendizaje potencialmente elevada) se vé notablemente aliviada con este enfoque.

Una ventaja adicional de los DSELs es la facilidad de integración con otros DSELs con el mismo lenguaje base. Este es un punto particularmente importante, ya que un DSL frecuentemente no se utiliza en forma aislada sino en colaboración con otros (por ejemplo, HTML suele utilizarse en combinación con un lenguaje de *scripting*, etc.). La integración de lenguajes siempre acarrea el riesgo de un desfasaje conceptual (un ejemplo típico de desfasaje conceptual ocurre cuando los lenguajes integrados adoptan diferentes paradigmas); la utilización de un mismo lenguaje base implica la adopción de un paradigma común, con lo cual el riesgo de desfasaje desaparece casi por completo.

Es interesante comentar que las ideas generales del enfoque DSEL existen desde hace bastante tiempo. En su paper clásico [Ben86], Jon Bentley comenta la idea de pensar a los lenguajes de programación en términos un núcleo básico, independiente de cualquier dominio de aplicación, sobre el cual se construyen vocabularios específicos a dominios particulares. Salvo diferencias menores, esta es la idea de DSEL adoptada hoy en día.

3.4.2 Elección de un Lenguaje Base

La elección de un lenguaje base no es una tarea trivial. Debido a que el DSEL heredará toda la infraestructura sintáctica y semántica del lenguaje en el que sea embebido, es importante que la misma sea flexible y adecuada para el dominio en cuestión. Existen muchos otros aspectos de variada importancia que también influyen en la elección, como la disponibilidad de herramientas de soporte (compiladores, intérpretes, etc.), la existencia de otros DSELs embebidos en el mismo lenguaje e incluso la difusión del lenguaje (utilizar un lenguaje muy difundido tiene la ventaja de que muchos usuarios lo conocerán, facilitando su aprendizaje del DSL).

Existen dos formas básicas de embeber un DSL; la primera es modificar la definición del lenguaje para adaptarse al dominio y la otra es extender al lenguaje. La primer forma es claramente más costosa, ya que implica reescribir muchas de las herramientas que en teoría podían utilizarse (como los compiladores). La segunda es más conveniente en cuanto a costos, pero el lenguaje base debe ser suficientemente flexible como para permitir la incorporación de nuevas características al mismo

nivel que las características primitivas. El segundo enfoque se suele denominar *embedding directo* y la forma más común de implementarlo es mediante bibliotecas de abstracciones (tipos, funciones, etc.) específicos al dominio.

En general, las características deseadas para el lenguaje base dependen del dominio del DSEL. Mientras que un lenguaje orientado a la descripción de especificaciones se verá beneficiado de la adopción de una semántica declarativa (y por lo tanto, se implementará mejor al ser embebido en un lenguaje declarativo), otro tipo de lenguaje puede encontrar indiferente la adopción de un estilo imperativo u orientado a objetos. Independientemente del dominio, un aspecto casi siempre fundamental en la elección tiene que ver con la flexibilidad de la sintaxis del lenguaje base; la misma debe permitir expresar las abstracciones de forma cómoda para los expertos del dominio y sin recurrir a excesivas palabras específicas del lenguaje base.

3.5 Resumen

En este capítulo presentamos el enfoque de lenguajes específicos al dominio (DSLs) para el desarrollo de software. Explicamos también la metodología general a seguir en el desarrollo con DSLs. Luego, analizamos las ventajas y desventajas del enfoque, y vimos como estas últimas pueden solucionarse utilizando un DSEL.

Consideramos que la idea de DSEL es muy prometedora y existen varios dominios que pueden beneficiarse de la misma. Creemos que es posible aprovechar esta idea para desarrollar DSELs basados en principios sólidos de ingeniería de *software*; esto motivaría la construcción de programas en forma eficiente, pero a la vez metódica y basada en modelos relativamente formales.

Parte II

HyCom: Un DSEL para Hypermedia

Capítulo 4

HyCom

HyCom es un lenguaje específico al dominio de hypermedia. La motivación principal detrás de su diseño es cubrir los requerimientos más importantes para el desarrollo de aplicaciones de este dominio. HyCom se encuentra embebido en el lenguaje Haskell, siendo esta una decisión de diseño que determina los aspectos esenciales del lenguaje resultante.

En este capítulo presentamos el DSEL HyCom. Comenzamos resaltando las razones de la elección de Haskell como lenguaje base de HyCom. Luego mencionamos los principios de diseño involucrados en el desarrollo del lenguaje y mostramos su estructura general. Además presentamos los elementos del DSEL que modelizan sus aspectos básicos, los requerimientos de navegación y las interfaces del usuario. Finalmente, estudiamos los aspectos relacionados con la persistencia y la extensibilidad del modelo.

4.1 Haskell como Lenguaje Base

Un paso fundamental en el desarrollo de un DSEL consiste en la elección del lenguaje base. Los requisitos deseables de un lenguaje base fueron analizados en el Cap. 3. Por varias razones importantes, elegimos al lenguaje funcional Haskell [PJH99] como lenguaje base para HyCom. Podemos dividir este proceso de elección en varias fases, que analizaremos en detalle en esta sección.

4.1.1 Análisis Preliminar de Requisitos

Comenzamos el proceso de selección a partir de un análisis de los requisitos del lenguaje base. Tomando en cuenta el área de aplicación de HyCom, distinguimos tres requisitos fundamentales a cumplir por el lenguaje elegido.

En primer lugar, como la función principal de HyCom consiste en describir diseños, es deseable contar con un estilo declarativo para ello. El estilo declarativo permite al desarrollador pensar en términos de sus ideas antes que en la forma en que las mismas se llevan a cabo.

Un segundo punto tiene que ver con la metodología composicional que deseábamos tomar en HyCom. Este tipo de metodología se basa en la realización de aplicaciones a partir de la composición de partes más simples. El lenguaje a elegir debe proveer una sintaxis y semántica adecuadas para la representación de estos conceptos.

En tercer lugar, consideramos deseable contar con chequeo estático de tipos. Un lenguaje centrado en la descripción de diseños como HyCom puede beneficiarse de gran manera del tipado fuerte, ya que el mismo provee una forma de verificar automáticamente la correctitud de los modelos.

4.1.2 Decisión de Utilizar un Lenguaje Funcional

En base a los requisitos establecidos en la fase anterior, decidimos utilizar un lenguaje funcional puro como base para HyCom. Esta elección está basada también en decisiones fundamentadas.

En primer lugar, los lenguajes funcionales puros soportan ampliamente el estilo declarativo. Además, al no presentar efectos laterales, el código es conceptualmente más claro y fácil de comprender. Por otro lado, entre los lenguajes declarativos más modernos y con mayor soporte se encuentran los lenguajes funcionales actuales (esta razón es importante, porque el lenguaje base debe contar con suficiente soporte para justificar el uso de un DSEL).

La metodología composicional puede ser enfocada muy naturalmente mediante el concepto de combinadores y transformadores, de uso muy difundido en la comunidad funcional [CH98]. Es precisamente el concepto de funciones de alto orden, provisto por los lenguajes funcionales, el que permite la representación efectiva del concepto en cuestión.

Finalmente, todos los lenguajes funcionales modernos optan por el chequeo estático de tipos. Además, la mayoría de los lenguajes funcionales actuales cuentan con sofisticados sistemas de tipos que permiten trabajar con tipado fuerte sin perder la flexibilidad.

4.1.3 Elección de Haskell

Habiendo determinado la utilización de un lenguaje funcional puro para embeber a HyCom, decidimos utilizar Haskell [PJH99] como base. La elección de Haskell no es arbitraria y responde a decisiones muy importantes.

Por un lado, Haskell es el lenguaje funcional puro estándar en la comunidad, contando con implementaciones muy eficientes [GHC99] y soporte constante [Has99]. La familia de lenguajes funcionales basados en ML [Aug84] cuentan con una difusión y soporte similar a la de Haskell, pero consideramos el hecho de que son impuros (es decir, permiten efectos laterales) como una desventaja: debido a que HyCom es principalmente un lenguaje de descripción de modelos, creemos que los efectos laterales son nocivos al dificultar el razonamiento matemático. Por otro lado, opinamos que la sintaxis de ML es menos flexible que la de Haskell.

El uso de combinadores y transformadores está muy difundido en Haskell. Parte de su éxito radica en la flexibilidad sintáctica de este lenguaje; en Haskell podemos establecer el uso infijo, postfijo o prefijo de las funciones, además de definir la asociatividad y la precedencia. Estos detalles contribuyen a incrementar la practicidad del uso de combinadores y transformadores.

En tercer lugar, Haskell provee un sistema de tipos estático muy potente, que soporta polimorfismo y otras características atractivas. Un aspecto fundamental del sistema de tipos de Haskell es su gran flexibilidad (ver Apéndice A), un requisito importante en un lenguaje fuertemente tipado.

Finalmente, existen puntos adicionales que hacían atractiva la utilización de Haskell. Por un lado existen gran cantidad de DSELs embebidos en Haskell, muchos de los cuales resultan interesantes para utilizar en conjunción con HyCom (como el acceso a bases de datos [LM99] y CGI [vDM96]). Por otro lado, la experiencia en la utilización de Haskell para embeber DSELs ha mostrado muy buenos resultados hasta el momento [CHJ93, EH97, Hud96b].

4.2 Principios de Diseño

El principio de diseño más importante de HyCom es la utilización de una metodología composicional en el desarrollo de aplicaciones. Este tipo de metodología consiste en la realización de una aplicación a partir de la composición de elementos más simples; el enfoque permite dirigir el desarrollo en forma descendente o ascendente, y además promueve el reuso de elementos existentes.

Un ejemplo claro del uso de esta metodología puede ilustrarse con el diseño de una interfaz gráfica. Normalmente, las interfaces de este tipo cuentan con elementos básicos como botones, campos de edición de texto, imágenes, barras de desplazamiento, etc. Pueden obtenerse nuevos componente a partir de la combinación de los anteriores. Por ejemplo, combinando un campo de edición de texto junto con una barra de desplazamiento puede obtenerse un nuevo tipo de campo de edición de texto. A su vez, la interfaz en su totalidad se construye a partir de la composición de elementos como los mencionados. Una ilustración del proceso mencionado puede verse en la Fig. 4.1.

En HyCom la metodología composicional mencionada se representa a través de componentes, combinadores y transformadores. Un *componente* es la representación explícita dentro del lenguaje de un elemento básico, como lo es un botón en una interfaz gráfica. Un *combinador* es una regla que determina cómo un conjunto de componentes se combinan para formar un nuevo componente. Por su parte, un *transformador* es una regla que se aplica sobre un componente y determina la variación o agregado de cierta propiedad del componente, resultando en uno nuevo.

Un combinador se representa en HyCom como una función que abstrae una regla de combinación en particular. Una función de combinación toma como argumentos los componentes a combinar, dando como resultado un nuevo componente resultante de aplicar la regla en cuestión. Un ejemplo de regla de combinación

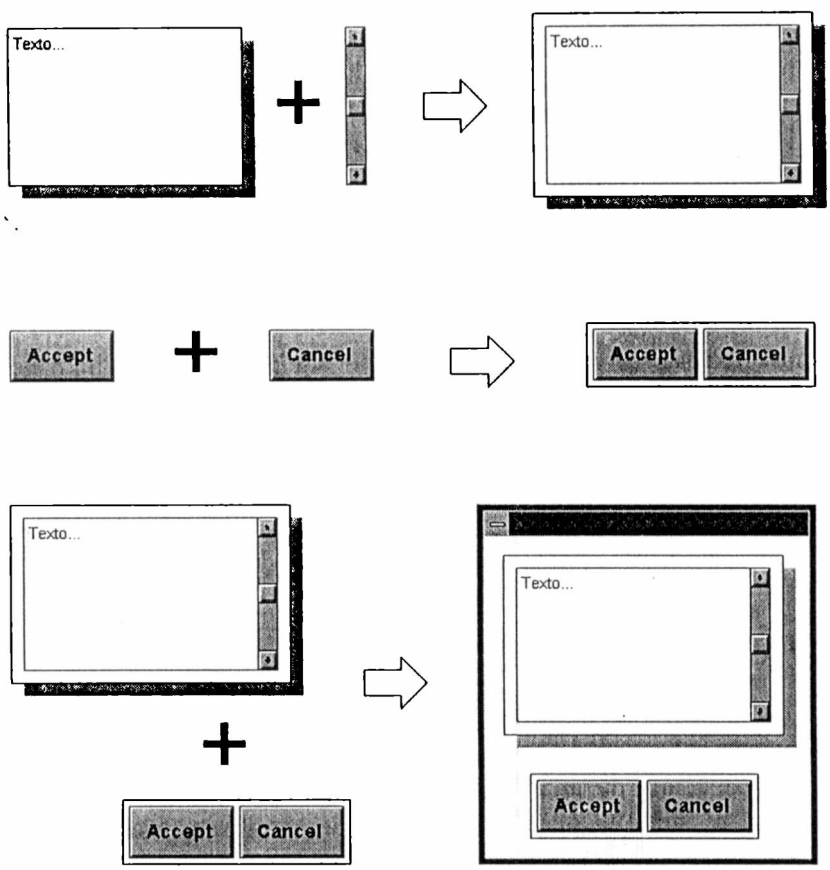

Figura 4.1: Ejemplo de Metodología Composicional

sería una regla de *layout* que determina que dos componentes compuestos de esta manera se visualizan uno arriba del otro. En HyCom, podemos representar esta regla mediante una función `/=\`, que al aplicarse a dos componentes `h1` y `h2`, nos da un nuevo componente resultante de ubicar `h1` encima de `h2` (ver la Fig. 4.2).

El concepto de transformador es representado en HyCom como una función que abstrae una regla de transformación en particular. Una función de transformación toma como argumento un componente, dando como resultado un nuevo componente resultante de aplicar la regla en cuestión. Un ejemplo claro de transformación puede verse también en el caso de una interfaz gráfica: suponiendo que disponemos de un componente de texto sin información de estilo, podemos transformarlo para obtener uno nuevo que tenga esa infomación. En HyCom, podemos representar esto mediante una función `textStyle` que toma como primer argumento el estilo deseado, como segundo argumento el componente de texto a transformar y da como resultado un nuevo componente con el estilo requerido (ver la Fig. 4.3).

En general, esta metodología basada en la combinación y transformación de componentes se utiliza en todo el desarrollo de una aplicación con HyCom. Se tie-

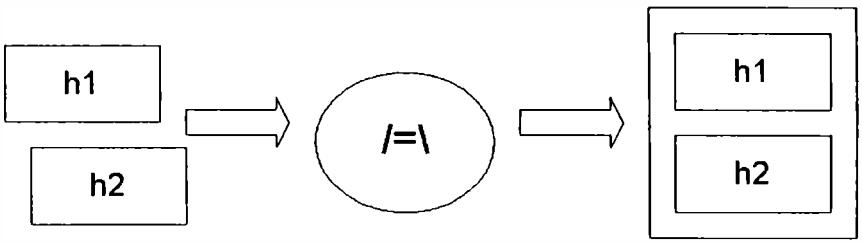

Figura 4.2: Ejemplo de un Combinador en HyCom

Figura 4.3: Ejemplo de un Transformador en HyCom

nen componentes cubriendo diferentes aspectos de la aplicación a desarrollar, que van desde la navegación por la hypermedia y la interfaz del usuario, hasta componentes representando los datos del dominio de aplicación particular. Se tendrán además combinadores y transformadores apropiados para cada uno de los aspectos mencionados.

Otro principio de diseño importante consiste en la utilización de clases de tipos para estructurar la arquitectura de HyCom. Una clase de tipos es un mecanismo que permite formalizar el compromiso de un componente de proveer ciertas operaciones (este mecanismo es diferente al concepto de clase del paradigma de orientación a objetos y se explica en detalle en el Apéndice A). Utilizando las mismas, podemos estructurar a los componentes en una jerarquía, dependiendo de las operaciones soportadas por los mismos.

El uso de las clases de tipos para representar las propiedades de un componente es comparable a la utilización de *interfaces* en el *desarrollo de software por componentes* [Szy98, COM99, Jav99]. En general, el desarrollo de *software* por componentes requiere algún tipo de mecanismo para especificar las responsabilidades y capacidades de un componente reusable. El reuso es efectivo si el desarrollador puede basarse únicamente en la interfaz de un componente y no en su funcionamiento interno; de esta forma, el proveedor del componente puede cambiar la representación interna sin que el desarrollador deba tomar recaudos al respecto.

En HyCom pueden verse algunos principios similares a los del desarrollo de *software* por componentes. Las clases son usadas para definir una estructura general basada en las responsabilidades de los componentes y no en su implementación. De esta forma, pueden existir componentes que tienen la misma responsabilidad pero implementada de diferente formas. El desarrollador que necesite un componente

con cierta responsabilidad puede elegir la implementación que más le convenga; ilustramos este concepto en la Fig. 4.4, en la cual vemos un combinador que puede aceptar más de una implementación de un componente que cumpla con cierta responsabilidad.

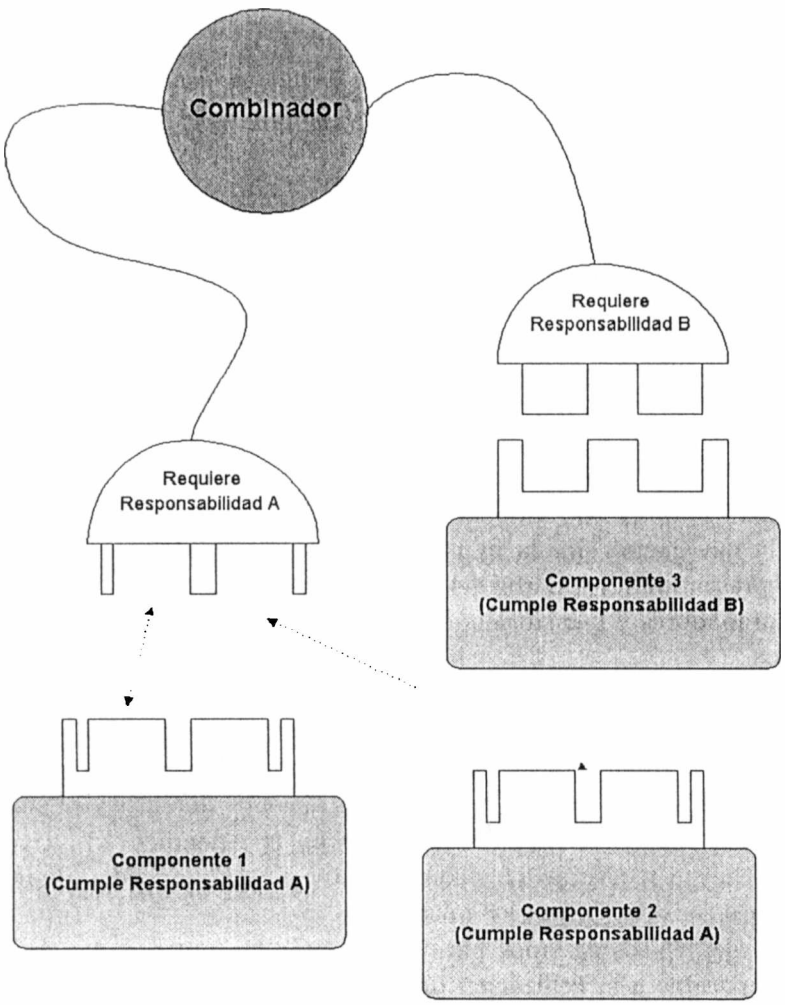

Figura 4.4: Componentes en HyCom

La utilización de clases de tipos no solamente permite representar elegantemente la estructura de HyCom, sino que facilita la extensibilidad del modelo. La incorporación de nuevas responsabilidades puede realizarse subclasificando las clases existentes. Por ejemplo, una clase representando un componente de interfaz de texto puede ser subclasificada para que permita mantener también el estilo del texto. De esta forma, las operaciones que requieran utilizar componentes de texto podrán utilizar aquellos que satisfagan los requerimientos de la clase original; las

operaciones que requieran manejar también el estilo de texto de sus componentes deberán especificar que requieren componentes de tipos que sean instancias de la subclase. Suponiendo que `HasText` y `HasStyle` son la clase y la subclase mencionadas respectivamente, el siguiente es un ejemplo de operaciones que requieren una u otra responsabilidad por parte de sus parámetros.

```
combineText :: (HasText a, HasText b) => a -> b -> ...
combineStyled :: (HasStyle a, HasStyle b) => a -> b -> ...
```

La extensibilidad también se manifiesta a nivel de la implementación de componentes. La creación de componentes que satisfagan ciertas responsabilidades se realiza a través de la instanciación de las clases apropiadas; en cualquier momento, un desarrollador puede extender el repertorio de componentes que realizan cierta responsabilidad sin necesidad de modificar el modelo base de HyCom.

4.3 Estructura General de HyCom

El DSEL HyCom está compuesto por varias partes, cubriendo diferentes aspectos del desarrollo de una aplicación hypermedial. Las distintas partes conforman varios grupos, distinguiéndose un núcleo que caracteriza al DSEL y partes accesorias que lo complementan (ver la Fig. 4.5). El núcleo es el que define los conceptos requeridos para caracterizar una aplicación hypermedial, como son la navegación y la interfaz del usuario. Las partes accesorias proveen algunas facilidades adicionales como el acceso a base de datos y la generación automática de prototipos.

El núcleo de la estructura de HyCom está dado por el DSEL propiamente dicho, siendo ésta la parte principal del lenguaje que hemos definido. Aquí es donde se implementan los principios fundamentales de diseño: la arquitectura de componentes, transformadores y combinadores, que define los requerimientos básicos de los mismos, estableciendo un modelo general sobre el cual se basan el resto de los elementos de HyCom.

La estructura general de una hypermedia es la primer capa de funcionalidad específica que extiende la estructura básica de componentes. Los requerimientos fundamentales de una hypermedia consisten en el soporte a la navegación y la interfaz del usuario. El núcleo del DSEL provee una arquitectura general mediante la cual puede proveerse soporte a estos conceptos. Complementando el núcleo hay varios módulos accesorios.

El complemento más importante está dado por el soporte para plataformas de implementación específicas. Su responsabilidad es la de proveer compiladores que traduzcan el diseño expresado en HyCom a una aplicación en una plàtaforma particular. Por ejemplo, el soporte a la plataforma de aplicaciones en la WWW permite compilar un diseño HyCom en un conjunto de páginas HTML. El soporte a plataformas no se limita solamente a la compilación de diseños en la plataforma en cuestión, sino también a proveer acceso a características propias de la misma (por ejemplo, el mecanismo de CGI en la WWW). El hecho de no incluir estas

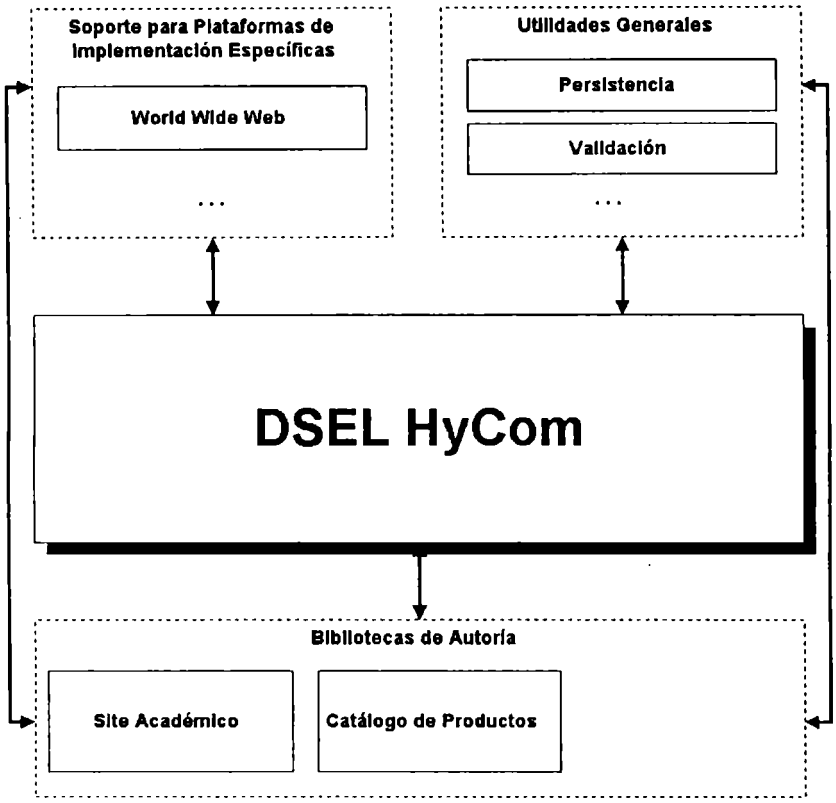

Figura 4.5: Estructura General de HyCom

características en el núcleo nos permite preservar su esencia independiente de plataforma.

Otra parte complementaria consiste en un conjunto de utilidades generales para el desarrollo (por ejemplo, los mecanismos de interacción con manejadores de bases de datos). Si bien no constituyen elementos fundamentales en la estructura de una hypermedia, en general son de gran utilidad en su implementación.

Finalmente, puede encontrarse la parte constituída por las bibliotecas de autoría. Una biblioteca de autoría consiste en una arquitectura genérica que captura un dominio de aplicación hypermedial determinado (por ejemplo, el dominio de los sistemas de información académicos). La arquitectura consiste en un modelo conceptual del dominio, un modelo de navegación definido sobre el modelo conceptual y una estructura general de interfaz del usuario. Utilizando una biblioteca de autoría el autor reusa la arquitectura definida para desarrollar su aplicación. De esta manera pueden reutilizarse tanto decisiones de diseño como componentes que son de frecuente aparición en el dominio, reduciendo considerablemente los tiempos de desarrollo.

4.4 Estructura del DSEL

La estructura del DSEL HyCom define la arquitectura básica que soporta los conceptos de componentes, combinadores y transformadores. Sobre esta arquitectura construimos la estructura general de soporte a la navegación y la interfaz del usuario.

El concepto fundamental en la arquitectura de HyCom es el de componente. Para el DSEL HyCom decidimos que la funcionalidad básica de un componente es la de poder ser identificado unívocamente. La funcionalidad es requerida debido a que todo componente en HyCom es potencial origen o destino de una actividad de navegación, con lo cual requiere poder ser referenciado en forma única. La identificación de un componente se lleva a cabo mediante un identificador, capturado dentro de HyCom mediante un tipo abstracto de datos, `ID`. Las operaciones soportadas por el tipo `ID` son la de creación y la de comparación por igualdad.

La responsabilidad de un componente es la de poder mantener un identificador. Determinamos a partir de esto que, dentro de HyCom, un componente es un elemento que soporta las operaciones de obtención y modificación de su identificador. Esta responsabilidad es formalizada mediante la clase de tipos `Component`.

```
class Component a where
   getID :: a -> ID
   setID :: ID -> a -> a
```

Para terminar de completar la arquitectura del DSEL definimos la estructura general de los transformadores y combinadores. La combinación básica provista por la arquitectura de HyCom es binaria, siendo la estructura más simple de combinación; combinaciones más complejas pueden obtenerse a partir de sucesivas combinaciones binarias. Una combinación es un componente formado por dos componentes más simples, junto con un elemento que representa la regla de combinación en particular.

```
data (Component a, Component b)
   => Combined c a b = Combined ID c a b
```

El tipo `(Combined c a b)` aloja dos componentes de tipos `a` y `b`, un elemento de tipo `c` y un `ID`. Mediante un contexto apropiado se establecen los requerimientos de que `a` y `b` sean componentes. Esto determina la restricción de que toda combinación está restringida a la composición de componentes. El elemento de tipo `c` representa la regla de combinación particular. Debido a que una combinación es un componente, debe realizarse la correspondiente instanciación de la clase `Component`. El `ID` mantenido por el tipo es manejado por las operaciones requeridas por esta clase.

La representación de una transformación sigue las mismas pautas adoptadas para las combinaciones. Una transformación es modelada, por lo tanto, mediante un componente que aloja al componente transformado junto con información sobre la regla de transformación.

```
data (Component a)
   => Transformed t a = Transformed ID t a
```

La estructura es similar a la de la combinación, en donde a es el tipo del componente transformado y `t` es el de la regla de transformación. Además, se mantiene un `ID` que es utilizado en la instanciación de la clase `Component`.

4.5 Componentes del Modelo Navegacional

El concepto fundamental en el acceso hypermedial a la información radica en la actividad de navegación. Un modelo navegacional provee soporte a esta actividad a través de una representación de los elementos que forman parte de la misma. En esta sección presentamos la estructura básica del modelo navegacional soportado por HyCom (ver la Fig. 4.6).

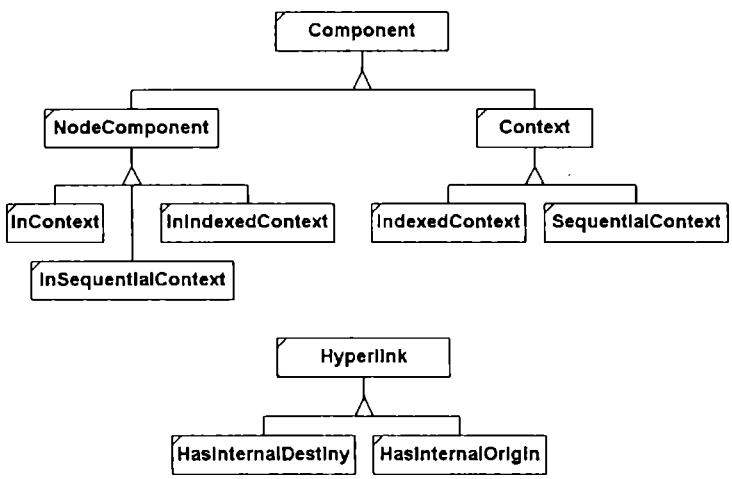

Figura 4.6: Diagrama de Clases de Tipos del Modelo Navegacional

La actividad de navegación consiste en recorrer unidades de información a través de estructuras de acceso apropiadas. Los nodos son unidades de información interconectadas por estructuras de acceso denominadas *links*. Esta estructura se describe en detalle en el Cap. 2.

El resultado que se obtiene al realizar el diseño navegacional de una aplicación es un modelo que describe la estructura navegacional de la misma, pero sin especificar cómo debe ser visualizado en la interfaz.

4.5.1 Nodos

En HyCom modelizamos el concepto de nodo mediante un componente. La necesidad de utilizar un componente es clara para el caso de un nodo, que debe poder ser identificado unívocamente como el destino u origen de un determinado *link*. Un

nodo además tiene otras responsabilidades asociadas, relacionadas con el hecho de ser origen o destino de *links*. Para representar estas responsabilidades adicionales subclasificamos la clase `Component` creando la clase `NodeComponent`; la clase hereda las responsabilidades de todo componente y agrega responsabilidades propias de los nodos.

En la modelización de un nodo en HyCom consideramos que su responsabilidad básica es la de conocer los *links* que emergen de él. Otras responsabilidades derivadas, como conocer sus nodos vecinos, pueden satisfacerse a partir de la anterior. Estas responsabilidades se encuentran dadas a través de las operaciones definidas en la clase `NodeComponent`.

```
class (Component a) => NodeComponent a where
   getOutgoingLinks  :: a -> Links
   getNeighbourhood  :: a -> Nodes
```

La operación `getOutgoingLinks` determina cuál es el conjunto de *links* que emergen a partir del nodo en cuetión. Por su parte, la operación `getNeighbourhood` indica el conjunto de nodos vecinos, los cuales constituyen los nodos destinos de los *links* que emergen a partir del nodo. El constructor `Links` hace uso de la cuantificación existencial (ver el Apéndice A) permitiendo que `getOutgoingLinks` pueda retornar cualquier tipo de *link*. De forma análoga, el constructor `Nodes` permite que `getNeighbourhood` pueda retornar cualquier tipo de nodo.

Los tipos `Links` y `Nodes` utilizan cuantificación existencial (ver Apéndice A), pudiendo contener a cualquier componente de la clase `Hyperlink` y `NodeComponent` respectivamente.

4.5.2 Links

Los *links* proveen un medio de interconectividad entre unidades de información; las unidades pueden ser nodos, o incluso unidades de granularidad más fina contenidas dentro de los nodos. En HyCom modelizamos los *links* como relaciones de navegación entre componentes; los componentes son la mínima unidad que puede ser origen o destino de un *link*.

Comenzamos el proceso de modelado de los *links* a partir del análisis de sus responsabilidades básicas. Decidimos que la responsabilidad básica de un *link* es la de vincular un nodo origen con un nodo destino. Siguiendo nuestro criterio de diseño habitual, modelizamos estas responsabilidades mediante una clase de tipos. Por lo tanto, las operaciones definidas para la clase `Hyperlink` son la de obtener los identificadores de los nodos origen y destino.

```
class Hyperlink a where
   getOriginNode  :: a -> ID
   getDestinyNode :: a -> ID
```

Existen *links* más específicos que vinculan elementos de granularidad más fina contenidos en los nodos. Decidimos modelizar esto a través de la especialización de

la clase `Hyperlink` en dos subclases. Subclasificamos la clase `Hyperlink` creando las clases `HasInternalDestiny` y `HasInternalOrigin`.

La clase `HasInternalDestiny` hereda las responsabilidades de `Hyperlink` y a su vez especifica la responsabilidad de conocer un componente específico dentro del nodo destino. Dicho componente representa precisamente el destino del *link*.

```
class (Hyperlink a) => HasInternalDestiny a where
   getInternalDestiny :: a -> ID
```

La clase `HasInternalOrigin` agrega la responsabilidad de conocer un componente específico dentro del nodo origen, el cual es el origen del *link*, de manera similar a `HasInternalDestiny`.

```
class (Hyperlink a) => HasInternalOrigin a where
   getInternalOrigin :: a -> ID
```

4.5.3 Contextos

La forma básica de especificar el modelo navegacional de una aplicación hypermedial se realiza a partir de nodos y *links*. Sin embargo, existen métodos avanzados que se utilizan para especificar características navegacionales de una aplicación, por ejemplo, mediante contextos navegacionales [SB94]. Los contextos se aplican sobre un conjunto de nodos y permiten al usuario realizar una navegación dentro de la hypermedia de forma controlada y consistente. Los contextos navegacionales han sido reconocidos como una forma de especificar la navegación en un nivel más alto que los nodos y *links*; forman parte de metodologías como OOHDM y han sido especificados también como pattern de hypermedia [RSG97].

El concepto de contextos se encuentra presente en HyCom, modelizado a partir de diferentes clases de tipos. El concepto básico se encuentra representado a través de la clase `Context`, mientras que contextos más específicos se representan especializando la clase anterior con las clases `IndexedContext` y `SequentialContext`.

En HyCom un contexto es un componente que agrupa varios nodos de acuerdo a una política particular (modelizado mediante la operación `getNodesInContext`). Un contexto tiene además una política de transformación de identificadores (modelizada con la operación `inContextNodeID`), de forma que todos los nodos presentes en un contexto ven modificado su identificador de la misma manera. La transformación de identificadores permite diferenciar el mismo nodo en diferentes contextos y realizar *hyperlinking* en forma acorde.

```
class (Component c) => Context c where
   inContextNodeID       :: c -> ID -> ID
   getNodesInContext     :: c -> [ID]
   ...
```

Un tipo de contexto más específico lo constituyen aquellos contextos que poseen un índice de los nodos que lo conforman, permitiendo el acceso a los mismos a través

de él. Este tipo de contexto es conocido como *contexto indexado*, y lo modelizamos en HyCom especializando la clase `Context` con la clase `IndexedContext`, cuya responsabilidad adicional es la de indicar el índice asociado al contexto.

```
class (Context c) => IndexedContext c s where
   contextIndex :: c -> ContextIndex s
```

La clase `IndexedContext` es *multi-parameter* (ver Apéndice A). El primer parámetro, `c`, representa el tipo del contexto mientras que el segundo, `s`, es el tipo de la clave asociada a cada entrada del índice. El tipo `(ContextIndex s)` mantiene el índice del contexto formado por los `IDs` de los nodos y por las claves de tipo `s`.

```
type ContextIndex s = [(s, ID)]
```

Otro tipo de contexto más específico lo constituyen aquellos contextos que determinan que sus nodos se encuentran ordenados secuencialmente, de manera tal que para cada nodo se tiene un nodo anterior y un nodo siguiente. Este tipo de contexto es conocido como *contexto secuencial*, y lo modelizamos en HyCom especializando la clase `Context` con la clase `SequentialContext`; la responsabilidad agregada en `SequentialContext` es la de poder determinar los nodos previo y siguiente de un nodo del contexto. La política de nodo anterior y nodo siguiente es determinada internamente por el componente que implementa el contexto.

```
class (Context c) => SequentialContext c where
   nextInContext     :: NodeComponent n => c -> n -> Maybe ID
   previousInContext :: NodeComponent n => c -> n -> Maybe ID
   ...
```

Para representar la propiedad de que un nodo pertenece a un contexto particular, especializamos la clase `NodeComponent` con la clase `InContext` que permite acceder a las características del contexto a partir de un nodo.

```
class (NodeComponent n) => InContext n where
   getContexts :: n -> [ID]
   ...
```

De manera semejante, para representar la propiedad de que un nodo pertenece a un contexto indexado o a un contexto secuencial, creamos en HyCom las clases `InIndexedContext` e `InSequentialContext`. Estas clases extienden a `NodeComponent` para permitir recuperar el índice del contexto a partir de un nodo, y permitir acceder a la funcionalidad del contexto secuencial a partir de un nodo, respectivamente.

4.6 Componentes de Interfaz del Usuario

El diseño de la interfaz del usuario es de importancia fundamental en el desarrollo de una aplicación hypermedial. En el diseño de la interfaz se identifican los elementos que el usuario percibirá al utilizar la aplicación; se determinan los elementos visuales que se utilizarán para mostrar los elementos del modelo navegacional. En esta sección, presentamos la estructura básica del modelo de la interfaz del usuario provista por HyCom (ver la Fig. 4.7).

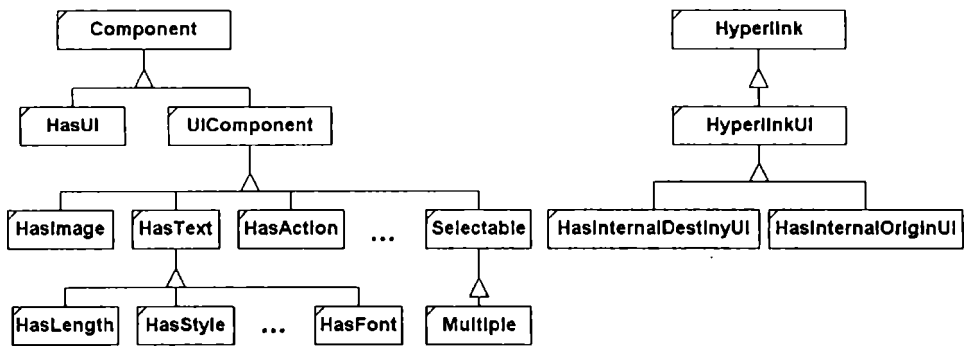

Figura 4.7: Interfaz del Usuario de HyCom

4.6.1 Componente de UI Básico

En HyCom modelizamos el concepto de un elemento de interfaz del usuario mediante un componente. De la misma manera como ocurría con los nodos en el modelo navegacional, la necesidad de utilizar un componente es clara: todo elemento de interfaz debe poder ser identificado unívocamente como el destino u origen concreto de una actividad de navegación (es decir, la representación visual del origen o destino de un *link*). Un elemento de interfaz posee otras responsabilidades además de la que posee todo componente. Para representar las responsabilidades adicionales subclasificamos la clase `Component` creando la clase `UIComponent`. La clase definida hereda las responsabilidades de todo componente y agrega responsabilidades propias de los elementos de interfaz.

En la modelización de un componente de interfaz del usuario en HyCom consideramos que las responsabilidades básicas de un componente de interfaz del usuario son la de mantener información con respecto al color de frente y fondo; las responsabilidades se encuentran dadas a través de las operaciones definidas en la clase `UIComponent`.

```
class (Component a) => UIComponent a where
   getBackground :: a -> Color
   getForeground :: a -> Color
   setBackground :: Color -> a -> a
```

```
setForeground :: Color -> a -> a
```

4.6.2 Especialización del Componente de UI Básico

Mientras que la clase `UIComponent` representa la responsabilidad básica de todo elemento de interfaz, existen diferentes elementos de interfaz que poseen responsabilidades adicionales (por ejemplo, mantener una imagen, mantener un texto, tener un determinado tamaño, tener asociada una determinada acción, etc.). Decidimos modelizar cada una de estas responsabilidades como una especialización de la clase `UIComponent`. A continuación presentamos alguna de las especializaciones; una descripción detallada se encuentra en el Apéndice B.

Existen variados componentes de interfaz que poseen un texto; ejemplo de esto son las etiquetas, los campos de texto, las áreas de texto y los botones. Especializamos la clase `UIComponent` con la clase `HasText` para poder especificar las responsabilidades adicionales de obtener e indicar el texto asociado a un elemento de interfaz.

```
class (UIComponent a) => HasText a where
   getText :: a -> String
   setText :: String -> a -> a
```

Otra responsabilidad adicional que poseen algunos componentes de interfaz es la de poder realizar una selección entre un conjunto de opciones posibles; en HyCom se lo modeliza con la clase `Selectable`. Como es fácil de ver, las operaciones asignadas para la clase definida son la de obtener e indicar el conjunto de opciones posibles.

```
class (UIComponent a) => Selectable a where
   getOptions :: a -> [Option]
   setOptions :: [Option] -> a -> a
```

Cada opción se encuentra modelizada con el tipo algebraico `Option`, el cual posee un determinado valor y una indicación de si el mismo se encuentra seleccionado o no.

```
data Option   = Option Selected Value
type Selected = Bool
type Value    = String
```

La clase `Selectable` puede estar especializada para poder indicar la responsabilidad de que en un componente se puedan hacer múltiples selecciones. Por ejemplo, el componente `listBox` definido en HyCom representa un conjunto de posibles opciones en donde puede definir que se realicen múltiples selecciones.

4.6.3 Transformadores y Combinadores de UI

Como se mencionó en la Secc. 4.2, el principio de diseño más importante en HyCom es el uso de una metodología composicional en el desarrollo de aplicaciones. La metodología determina que una aplicación se realiza a partir de la composición de elementos más simples.

Al desarrollarse una interfaz se poseen diferentes componentes de interfaz básicos, como botones, imágenes, campos de edición de texto, barras de desplazamiento, etc. Componentes más complejos se obtienen a partir de la combinación de los componentes básicos. A su vez, la interfaz de una aplicación se encuentra constituída a partir de la composición de los elementos mencionados.

En HyCom se encuentran definidos varios transformadores y combinadores de componentes de interfaz básicos. Por ejemplo, hemos definido un combinador que permite especificar la disposición gráfica entre dos componentes de interfaz. La combinación se encuentra definida a partir del tipo `Layout`, permitiendo especificar que un componente se ubica por encima del otro, o que un componente se ubica al costado del otro, o simplemente que uno se ubica a continuación del otro.

```
data Layout' = AboveOf_Left | AboveOf_Center | AboveOf_Right |
               LeftOf_Top   | LeftOf_Center  | LeftOf_Bottom |
               Next

type Layout a b = Combined Layout' a b
```

Definimos funciones para poder especificar las combinaciones relacionadas a la disposición gráfica de dos componentes. Por ejemplo, el combinador `(/=\)` especifica que el primer componente se encuentra ubicado encima del segundo, además de alinearlos en forma centrada.

```
aboveOf_Center, (/=\) :: (UIComponent a, UIComponent b)
                            => a -> b -> Layout a b
aboveOf_Center a b     = Combined (getID a) AboveOf_Center a b
(/=\)                  = aboveOf_Center
```

De manera semejante, el combinador `(<=<)` ubica el primer componente a la izquierda del segundo, también disponiéndolos en forma centrada. Los restantes combinadores de *layout* se definen de forma similar.

```
leftOf_Center, (<=<)  :: (UIComponent a, UIComponent b)
                            => a -> b -> Layout a b
leftOf_Center a b      = Combined (getID a) LeftOf_Center a b
(<=<)                  = leftOf_Center
```

Uno de los transformadores básicos que se encuentran en HyCom es `AnchoredUI`, que permite "anclar" un componente con un determinado link.

```
data (Hyperlink l) => SimpleAnchorUI l = SimpleAnchorUI l
type AnchoredUI l a = Transformed (SimpleAnchorUI l) a
```

Definimos la función anchoredUI para poder aplicar la transformación correspondiente.

```
anchoredUI    :: (Hyperlink l, UIComponent a)
                  => l -> a -> AnchoredUI l a
anchoredUI l a = Transformed (getID a) (SimpleAnchorUI l) a
```

4.6.4 Asociar un UI a un Componente o Link

Cada uno de los componentes definidos en HyCom puede tener una interfaz de usuario asociada. Definimos la clase de tipos HasUI para permitir especificar la propiedad de que un componente tiene una representación de interfaz asociada.

```
class (Component a) => HasUI a where
   getUI :: a -> ID
```

Un componente que es instancia de esta clase puede determinar cuál es el identificador del componente de interfaz que lo representa.

En HyCom se encuentra definida la clase HyperlinkUI para poder determinar cuáles son las representaciones de interfaz de los nodos origen y destino de un *link*. La clase permite determinar la representación de una relación de navegación en la interfaz.

```
class (Hyperlink a) => HyperlinkUI a where
   getOriginNodeUI  :: a -> ID
   getDestinyNodeUI :: a -> ID
```

Especializamos la clase HyperlinkUI con la clase HasInternalDestinyUI para agregar la responsabilidad de que un link con un destino interno a un nodo indique cuál es el componente de interfaz del usuario asociado.

```
class (HyperlinkUI a) => HasInternalDestinyUI a where
   getInternalDestinyUI :: a -> ID
```

Existe una clase HasInternalOriginUI que permite especificar el origen interno de un *link* y se define de forma análoga.

4.7 Persistencia de Componentes

Habitualmente, las aplicaciones hypermedia mantienen los datos de su modelo conceptual o navegacional almacenados en una base de datos. Incluso, dependiendo del tipo de aplicación, muchas de ellas almacenan datos durante su ejecución (posiblemente, datos correspondientes a los usuarios). En cualquier caso, la posibilidad de recuperar y almacenar datos en almacenamiento secundario es muy importante en una aplicación.

HyCom provee manejo de persistencia de componentes en forma muy elegante, permitiendo la interacción con diferentes medios de almacenamiento de manera uniforme. A lo largo de esta sección vemos los conceptos fundamentales involucrados en el manejo de persistencia con HyCom.

4.7.1 Retriever y Storer en HyCom

HyCom provee una forma de interactuar con medios de almacenamiento en forma flexible y elegante a través de los conceptos de *retriever* y *storer*. Un *retriever* es un elemento que permite recuperar ciertos tipos de componentes de algún medio de almacenamiento (base de datos, archivos, etc.). De forma análoga, un *storer* permite almacenar componentes en algún medio de almacenamiento. Estos conceptos se representan en HyCom mediante clases *multi-parameter*.

Definimos en HyCom la clase `Retriever` para modelizar la responsabilidad de poder recuperar cierto tipo de componente de algún medio de almacenamiento determinado.

```
class Retriever media component where
   retrieve   :: media -> IO [component]
   retrieveBy :: media -> (component -> Bool) -> IO [component]
```

La operación `retrieve` recupera todos los componentes de tipo `component` en el medio de almacenamiento indicado. Por su lado, la operación `retrieveBy` recupera sólo los componentes que cumplen el predicado especificado.

Por otro lado, la funcionalidad de almacenar componentes en un cierto medio de almacenamiento se encuentra modelizado en HyCom mediante la clase `Storer`.

```
class Storer media component where
   store :: media -> [component] -> IO()
```

La operación `store` almacena la lista de componentes en el medio de almacenamiento indicado.

Las clases `Retriever` y `Storer` pueden subclasificarse, llevando a *retrievers* y *storers* más complejos. Por ejemplo, podríamos definir clases para *retrievers* exclusivamente sobre bases de datos relacionales, y que por lo tanto podrían tomar una consulta SQL y realizar la misma.

4.7.2 Retriever y Storer para HaskellDB

Al utilizar una base de datos como medio de almacenamiento de datos, es usual que las consultas sean comunicadas a la misma como *strings* no estructurados representando expresiones SQL. La problemática asociada es que se posee un enfoque de bajo nivel, provocando varias desventajas.

El DSEL HaskellDB [LM99] permite expresar consultas y otras operaciones sobre bases de datos relacionales de una manera segura (en cuanto a chequeo de tipos) y declarativa. Todas las tablas, consultas y operaciones son expresadas completamente en Haskell, sin la necesidad de comandos SQL.

A modo de ejemplo, vemos cómo podemos expresar una tabla y consultas en HaskellDB. La siguiente expresión define una tabla de autores utilizando los tipos `Table` (representando una tabla) y `Expr` (representando un atributo) provistos por el DSEL.

```
authors :: Table ( address  :: Expr String,
                   au_fname :: Expr String,
                   au_id    :: Expr String,
                   au_lname :: Expr String,
                   city     :: Expr String )
```

A pesar de que a la base de datos se envían *strings* de código SQL, las consultas son expresadas con funciones Haskell. El DSEL provee un conjunto de funciones de manipulación de datos, como `restrict` (análoga a la operación de selección en SQL) y `project` (análoga a la operación de proyección). Estas operaciones tienen la ventaja adicional del chequeo estático de tipos.

A modo de ejemplo definimos una consulta que determina los autores que viven en la ciudad de Buenos Aires. La consulta está expresada en código monádico (ver Apéndice A) y utiliza las funciones `restrict` y `project`. Aclaramos que el operador `(!)` permite obtener el atributo de una tupla.

```
enBuenosAires
        = do{ x <- table authors
            ; restrict (x!city .==. constant "Buenos Aires")
            ; project (au_fname = x!au_fname
                      ,au_lname = x!au_lname)
            }
```

El ejemplo se encuentra simplificado y hace omisión de los detalles de implementación muy específicos. A pesar de eso nos muestra cuál es el estilo general de especificar tablas y consultas en HaskellDB.

Debido a que tanto HyCom como HaskellDB son DSELs embebidos en Haskell su integración es prácticamente inmediata. El usuario puede definir *retrievers* y *storers* que utilicen bases de datos relacionales mediante este DSEL.

El siguiente ejemplo muestra la instanciación de un *retriever* utilizando HaskellDB. La definición utiliza una función `retrieveFromDB` (provista por HyCom), que toma la referencia a la base de datos (especificada mediante un DSN, por ser una base de datos ODBC [ODB99]), un *parser* de tablas `authorParser` (que permite obtener una componente de tipo `Author` a partir de una tupla) y el query `authorQuery` (que recupera todos los autores de la base de datos).

```
instance Retriever BD_Retriever Author where
   retrieve t = retrieveFromDB (retrieverDSN t)
                               authorParser
                               authorQuery
```

Existen detalles técnicos adicionales en la instanciación que omitimos para preservar la claridad del ejemplo.

4.8 Extensibilidad del Modelo

HyCom provee una arquitectura general donde basar las aplicaciones, permitiendo además una fácil extensibilidad del mismo para poder cubrir requisitos específicos de aplicaciones particulares. La arquitectura puede extenderse a partir de la subclasificación de las clases existentes. Las clases nuevas definirían la estructura que deben cumplir nuevos tipos de componentes. A continuación vemos algunos ejemplos que muestran cómo se puede extender la arquitectura base.

Un ejemplo simple de extensión de la estructura de clases es la inclusión de *annotated links*. Un *link* de este tipo, además de la información de asociación entre nodos, posee un *string* de texto explicativo. El mismo podría ser luego interpretado en la interfaz como una breve explicación que aparece al pasar el marcador del *mouse* encima del *anchor* correspondiente. La extensión al modelo implica definir la siguiente clase.

```
class (Hyperlink a) => AnnotatedHyperlink a where
   getAnnotation :: a -> String
   setAnnotation :: String -> a -> a
```

La clase `AnnotatedHyperlink` hereda los requerimientos de operaciones de la clase `Hyperlink` y agrega el requerimiento de implementar las operaciones para manejar la anotación. Un componente que implemente esta funcionalidad deberá implementar las operaciones prescriptas por la clase `Hyperlink` y también las operaciones `getAnnotation` y `setAnnotation`.

Otro ejemplo es la definición de nodos específicos a una aplicación en particular. La clase `NodeComponent` define la funcionalidad básica de un nodo, pero es probable que en un dominio de aplicación particular un nodo tenga características propias de ese dominio. Por ejemplo, en una aplicación académica es factible tener una clase de nodo que mantenga la información de un profesor.

```
class (NodeComponent p) => ProfessorNode p where
   toCoursesGiven :: p -> [StringBasedAnchor]
   toDepartments  :: p -> [StringBasedAnchor]
```

La definición permite pedir al nodo los *links* que emergen de él ya que la nueva clase es subclase de `NodeComponent`. Sin embargo, permite también pedir *anchors* a otros nodos, siguiendo una semántica específica del dominio de la aplicación desarrollada. El tipo `StringBasedAnchor` es provisto por HyCom e implementa una versión simple de *anchor*.

Otra posible extensión sería especializar un *retriever* para que funcione con una base de datos relacional, tomando una consulta SQL para recuperar los datos. La especialización podría mejorar la *performance* del nuevo *retriever* sobre uno normal cuando se trabaje con bases de datos relacionales. Podemos definir la clase `SQLRetriever` para modelizar la funcionalidad descripta.

```
class (Retriever media c) => SQLRetriever media c where
   retrieveBySQL :: media -> Query -> IO [c]
```

Finalmente, vale comentar que cuando se define el modelo conceptual de una aplicación es muy probable que desee definirse clases que capturen los diferentes objetos conceptuales. La definición con clases es más flexible y genérica que la definición mediante tipos. Veremos esto mediante ejemplos en el Cap. 6.

Como se vé en esta sección, la arquitectura de HyCom permite *fácil extensibilidad* para cubrir requisitos específicos de aplicaciones particulares. La metodología general a seguir es derivar subclases especializando el comportamiento de las existentes de acuerdo a las necesidades de la aplicación.

4.9 Resumen

En este capítulo presentamos la estructura general del DSEL HyCom. HyCom toma como concepto fundamental el de componente, combinadores y transformadores. Este es un concepto de diseño muy utilizado que consiste en la construcción de aplicaciones a partir de la combinación de componentes simples para formar componentes más complejos.

La estructura de HyCom está dada por clases de tipos. Las clases de tipos permiten capturar diferentes propiedades de los distintos grupos de componentes. HyCom provee componentes cubriendo diferentes aspectos de la aplicación, como el diseño navegacional y el diseño de la interfaz. Los conceptos de *retriever* y *storer* integran HyCom con medios de almacenamiento de datos, permitiendo manejar la persistencia de los componentes. Finalmente, la jerarquía de HyCom puede ser extendida, permitiendo definir nuevas clases a partir de las existentes.

Capítulo 5

Plataformas y Compilación

Independientemente del método o modelo de diseño utilizado en el desarrollo de una aplicación, el producto final está muy influenciado por la fase de implementación. Cualquier herramienta de desarrollo que se utilice debe considerar en cierto momento la traducción del diseño a alguna plataforma de implementación (por ejemplo, la WWW para aplicaciones hypermediales). Por otro lado, si bien en general un diseño es independiente de plataforma, la inclusión de características específicas de la misma pueden ser importantes en el producto final.

En este capítulo analizamos de qué forma HyCom soporta la generación automática de prototipos y aplicaciones. Vemos además cómo pueden incluirse características específicas de una plataforma particular. Finalmente, vemos cómo se aplican estos principios en el desarrollo de un módulo de soporte para la plataforma de aplicaciones WWW.

5.1 Generación Automática de Prototipos

Uno de los puntos que restan aceptación a las metodologías de diseño es la excesiva separación entre el diseño y la implementación. Por un lado, esta separación puede ser causa de errores que se descubren en forma tardía (recién al tener el producto final) y que para entonces pueden ser difíciles de solucionar. Por otro lado, el tener que realizar la traducción de los elementos del diseño a la implementación en forma manual le resta eficiencia al desarrollo y aumenta su costo.

En el Cap. 2 vimos que una solución al problema de la evaluación tardía consiste en proveer una facilidad de prototipación automática. De esta forma, el diseño puede traducirse inmediatamente a la implementación con una mínima intervención del usuario. De la misma manera, puede utilizarse una facilidad similar para generar la aplicación final en forma automática. Teniendo tales facilidades podríamos enfocar el diseño en forma de prototipos evolutivos (o espiral).

5.2 Soporte de Plataformas de Implementación

Gran parte de los puntos que determinan la calidad del producto final quedan definidos en un buen diseño. Por ejemplo, la estructura del esquema de navegación, la correcta distribución del contenido de los nodos, etc. Sin embargo, es muy probable que existan detalles de implementación dependientes de la plataforma de la aplicación final que sean fundamentales para la aceptación del mismo.

Por lo tanto, si bien la independencia de plataforma es importante en las fases iniciales del desarrollo, reconocemos la importancia de poder soportar características específicas. El soporte de las mismas debe hacerse de forma transparente, de manera tal de no oscurecer el diseño ni establecer compromisos que afecten las primeras fases del desarrollo.

5.3 Plataformas y Compilación en HyCom

HyCom soporta la integración con plataformas específicas a través de bibliotecas a tal efecto. Los requerimientos de la integración con una plataforma son los siguientes.

- Proveer funciones de compilación que permitan obtener una aplicación funcional a partir de un diseño expresado en HyCom.
- Proveer una forma de utilizar características específicas de la plataforma en el marco provisto por HyCom.
- Permitir extensibilidad, de forma que el usuario final pueda proveer la forma de compilación de nuevos componentes sin necesidad de modificar la biblioteca base.

El principio de diseño de las bibliotecas de integración con plataformas es el mismo seguido a lo largo del diseño de HyCom. La propiedad que posee un componente de ser compilable bajo determinada plataforma es capturada por una clase de tipos. De esta manera, para una plataforma *X* se tendrá una clase de tipos de la siguiente forma.

```
class (Component a) => XCompilable a where
   compileToX :: a -> XApplication
```

La definición de la clase que mostramos aquí está simplificada por cuestiones de claridad. Es posible que la operación `compileToX` tome algún parámetro adicional (como información necesaria para la compilación, referida a cuestiones de *performance*, detalles de los nombres de archivo a generar, etc.). Además, es probable que la función deba utilizar efectos laterales (entrada/salida, etc.), con lo cual debería utilizarse `(IO XApplication)` como tipo del resultado.

En la definición anterior, el tipo `XApplication` es una representación abstracta de una aplicación en la plataforma *X*. Esta representación puede traducirse directamente a una aplicación funcional en la plataforma en cuestión.

Por cada componente que se desee hacer compilable bajo la plataforma *X* se tendrá una declaración de instancia. Supongamos, por ejemplo, que deseamos permitir que un `TextLabel` sea compilable en la plataforma *X*. La declaración tendrá la siguiente forma.

```
instance XCompilable TextLabel where
   compileToX t = ...
```

La definición de la operación `compileToX` determinará de qué forma se traduce el componente a la plataforma en cuestión.

Esta forma de diseñar la integración con plataformas nos permite definir qué componentes son compilables. Mientras utilicemos componentes compilables tendremos la posibilidad de generar prototipos y aplicaciones automáticamente para la plataforma en cuestión. El mecanismo permite extensibilidad, ya que para cada nuevo componente que se desee hacer compilable, basta una declaración de instancia asociada.

5.4 Una Plataforma para Desarrollo de Aplicaciones en la WWW

La WWW es hoy en día una de las plataformas más requeridas en el desarrollo de aplicaciones hypermedia. Por esta razón, elegimos la misma como primer plataforma a cubrir en las bibliotecas de integración. En esta sección veremos cómo los principios de integración con plataformas se siguen en el desarrollo de una biblioteca para integración con la WWW.

En la biblioteca de integración con la WWW se distinguen dos partes, producto del principio de diseño comentado en la sección anterior. Por un lado, proveemos un compilador HTML que permite traducir los diseños a un conjunto de páginas HTML. Por otro lado, proveemos componentes representando características específicas de la plataforma (como CGI, *Frames*, etc.).

5.4.1 El Compilador HTML

El compilador HTML de HyCom está formado por varios módulos. El proceso se divide básicamente en dos fases (ver la Fig. 5.1).

1. Una función de compilación toma un componente, junto con información de compilación, y genera un código intermedio denominado *Meta-HTML*.

2. Una función de *rendering* toma el código Meta-HTML generado y produce un conjunto de archivos HTML.

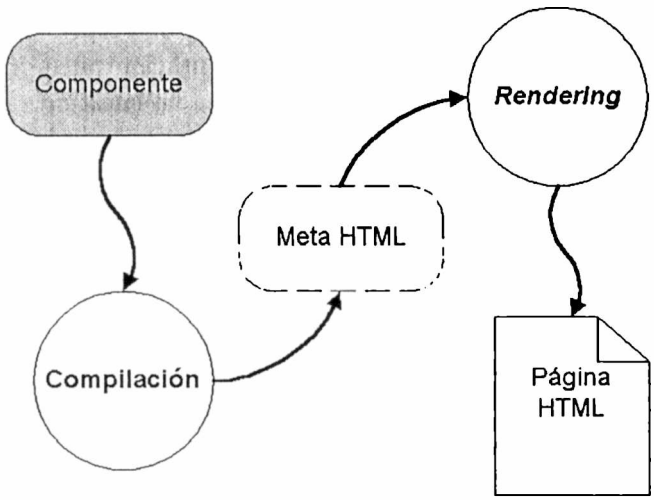

Figura 5.1: Estructura del Compilador HTML

El código Meta-HTML es una representación intermedia del código HTML más manejable que el HTML puro. Representamos este código mediante un tipo abstracto `MetaHTML`. La función `rawText` es la operación más simple sobre el tipo; permite crear un elemento `MetaHTML` a partir de un `String`.

```
rawText :: String -> MetaHTML
```

La operación `tagged` toma un elemento de tipo `MetaHTML` y un *tag* HTML (representado por el tipo abstracto `Tag` provisto por la biblioteca), y retorna un elemento `MetaHTML`.

```
tagged :: Tag -> MetaHTML -> MetaHTML
```

Por otro lado, la función `justTag` permite construir un elemento de tipo `MetaHTML` consistente en un único *tag*.

```
justTag :: Tag -> MetaHTML
```

Otra de las operaciones importantes consiste en el combinador secuencial de componentes `MetaHTML`, que permite secuenciar dos componentes del tipo retornando un nuevo componente.

```
(+-+) :: MetaHTML -> MetaHTML -> MetaHTML
```

La ventaja de utilizar el tipo abstracto `MetaHTML` consiste en independizar la compilación del código HTML generado finalmente. Esto no sólo permite abstraernos de eventuales cambios en el lenguaje HTML, sino también hacer transparente al usuario los cambios en el funcionamiento interno de ciertos aspectos del compilador.

El aspecto principal del compilador HTML está dado por las clases que determinan las propiedades de compilación. Esto sigue los principios esbozados en la sección anterior. La estructura del compilador HTML está dada básicamente por las siguientes clases.

```
class (Component a) => HTMLCompilable a where
   compileToHTML :: CompilationInfo -> a -> IO MetaHTML

class HTMLCompilableCombinator c where
   compileToHTMLC :: (Component a, Component b,
                      HTMLCompilable a, HTMLCompilable b) =>
                      CompilationInfo -> Combined c a b ->
                      IO MetaHTML

class HTMLCompilableTransformer t where
   compileToHTMLT :: (Component a, HTMLCompilable a) =>
                      CompilationInfo -> Transformed t a ->
                      IO MetaHTML
```

Las tres clases definidas determinan la propiedad de compilación de componentes, combinadores y transformadores respectivamente. Existen otras clases auxilares que no mostramos aquí por simplicidad. Las operaciones de las clases toman información adicional, contenida en el tipo `CompilationInfo`. Básicamente, esta información consiste en los nombres de los archivos a ser generados finalmente (estos nombres pueden ser generados automáticamente por HyCom o provistos por el usuario en el momento de la compilación).

Para hacer un componente compilable bajo HTML sólo es necesario realizar la instancia correspondiente al mismo. Por ejemplo, la siguiente es la instanciación del componente `Image`, que representa una imagen estática. Aclaramos que la función `tag` utilizada pertenece al tipo abstracto `Tag` y permite crear un elemento del tipo.

```
instance HTMLCompilable Image where
   compileToHTML ci (Image i _ _ s d)
        = return $ justTag (tag "IMG" [idToAttribute i,
                                       "SRC=" ++ s,
                                       "ALT=" ++ d,
                                       "BORDER=0"])
```

El proceso de generación de código HTML prosigue después de la fase de compilación en sí, pasando por una fase de *rendering*. Esta fase es mucho más directa y consiste en convertir el código Meta-HTML en un conjunto de archivos HTML. Existen varias funciones involucradas con esta conversión; la función `renderMetaHTML`, por ejemplo, construye un `String` con código HTML a partir de un elemento de tipo `MetaHTML`.

```
renderMetaHTML :: MetaHTML -> String
```

Es claro que el usuario no necesita ocuparse de esta fase, que es manejada internamente por HyCom. La única preocupación del usuario tiene que ver con la utilización de componentes compilables, o en todo caso, de proveer las definiciones apropiadas para que el componente pase a ser compilable bajo la plataforma elegida.

5.4.2 Características Específicas de HTML

La WWW tiene varias características que le son propias. La mayoría de las mismas representa alguna característica que también se encuentra en plataformas más complejas, pero que toma un carácter especial debido a las condiciones en las que funciona la WWW. Ejemplo de esto son los *frames*, una forma particular de lograr un efecto de múltiples ventanas superpuestas, y la interacción mediante CGI y *forms*, que permite tener aplicaciones interactivas con un modelo muy simple.

El módulo de integración con la WWW provee formas de utilizar estas características. Es importante poder proveer las mismas, debido a que como ya mencionamos, es probable que los requerimientos de la aplicación impliquen el uso de alguna de ellas. A continuación mostramos la forma en que se integran en el marco de HyCom.

Frames

Los *frames* son una forma bastante primitiva de mantener un efecto similar a múltiples ventanas abiertas en una página WWW. Los *frames* presentan muchas desventajas a la hora de la navegación, y son además bastante engorrosos de definir. Sin embargo, a veces es necesario recurrir a ellos para lograr determinados efectos en la interfaz. Deseamos por lo tanto poder representarlos lo más claramente posible dentro de HyCom.

Un *frame* está formado básicamente por dos elementos: un *frameset* y los *frames* en sí. El primero define un marco en el cual se insertan los diferentes *frames*, de forma horizontal o vertical. Los segundos son básicamente páginas que pueden contener código convencional u otros *framesets*. Para una explicación más detallada remitimos al lector a la especificación del lenguaje HTML [HTM99].

En HyCom vemos a los *frames* como una forma especial de elementos de interfaz (la diferencia con los componentes de interfaz radica en que no pueden combinarse con la misma versatilidad, debido a sus limitaciones). Representamos un *frame* con un elemento de tipo `Frame` que construímos a partir de un componente de interfaz convencional mediante la función `frame`.

```
frame :: (Component a) => a -> Frame
```

Los elementos de tipo `Frame` pueden insertarse en un *frameset* representado por el tipo `FrameSet`. Utilizamos la función `frameSet` para construir elementos de este tipo; la misma toma la lista de *frames*, junto con un elemento de tipo `FrameType` que indica si los *frames* se insertarán en forma horizontal o vertical.

```
frameSet :: [Frame] -> FrameType -> FrameSet
```

Un *frameset* puede contener a otros *framesets*. Para esto debemos construir un componente de tipo `Frame` a partir de un `FrameSet` utilizando la siguiente función.

```
nestedFrameSet :: FrameSet -> Frame
```

Existen otras operaciones para manejar *frames* en HyCom que abarcan detalles de presentación (como por ejemplo, el tamaño, las barras de desplazamiento, etc.) y no las mostramos aquí ya que no aportan elementos novedosos.

A modo de ejemplo vemos cómo definir un conjunto de *frames* con las herramientas provistas por HyCom. El elemento page0 define el *frameset* principal, que consiste en los cuatro *frames* compuestos verticalmente (ya que el segundo parámetro de `frameSet` es `VFrame`). Observemos también que el cuarto elemento insertado en el *frameset* principal es otro *frameset*.

```
page0 = setID (iD "Page0") $
          frameSet [frame page1,
                    frame page2,
                    frame page3,
                    nestedFrameSet auxFrame] VFrame

   where page1 = ...
         page2 = ...
         page3 = ...
         auxFrame = ...
```

La página resultante de la definición anterior puede verse en la Fig. 5.2.

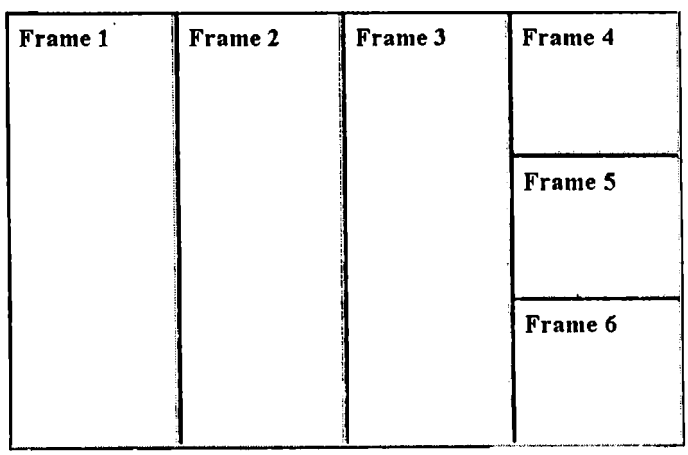

Figura 5.2: Ejemplo de *Frames*

Forms y CGI

Los *forms* y CGI son la primer forma que existió de proveer interacción en la WWW, y siguen siendo muy utilizados debido a su simplicidad y a que es una forma soportada por prácticamente todos los *servers*. HyCom se beneficia aquí de ser un DSEL, y se integra directamente con el DSEL para manejo de CGI *CgiLib* [vDM96].

La biblioteca *CgiLib* maneja la forma del pasaje de datos en forma transparente al usuario, quien sólo se limita a tomar los datos y retornar el *output* en HTML. La biblioteca provee una función `cgi` que toma el programa del usuario, de tipo `(Query -> IO HTML)`, y lo ejecuta.

```
cgi :: (Query -> IO HTML) -> IO ()
```

El tipo `Query` contiene la información pasada a través de CGI (por ejemplo, la información proveniente de los *forms*). El tipo `HTML` es una representación de HTML similar al `MetaHTML` provisto por HyCom. La presencia de la mónada `IO` (ver Apéndice A) nos indica que el programa del usuario puede involucrar efectos laterales.

La conexión de HyCom con la biblioteca de CGI resulta en una generalización de la función `cgi`, permitiendo devolver cualquier componente compilable en HTML como *output*.

```
hyComCGI :: HTMLCompilable a => (Query -> IO a) -> IO ()
```

La facilidad de invocar *scripts* CGI funciona en conjunción con los *forms*. Los *forms* proveen un conjunto de *widgets* de ingreso de datos (campos de edición de texto, botones, etc.). Un *form* además tiene una acción asociada (el programa CGI que debe invocarse). HyCom permite una representación de *forms* mediante componentes y clases que resulta muy elegante y clara conceptualmente; un componente de tipo `(Form a)` resulta de la transformación de un componente de interfaz de tipo a mediante la acción del *form* (representada por el tipo `FormAction`).

```
form :: (UIComponent a) => FormAction -> a -> Form a
```

Un *form* es entonces una transformación sobre una combinación de componentes de interfaz (que pueden incluir componentes de *input*, como editores de texto, etc.). La transformación asocia una acción con la combinación de componentes. Debido a que un *form* tiene una acción asociada, el mismo es instancia de la clase `HasAction`.

```
instance (UIComponent a) => HasAction (Form a) FormAction where
   getAction aForm           = ...
   setAction newAction aForm = ...
```

Otro aspecto importante de los *forms* es que frecuentemente son utilizados para mantener estado entre invocaciones CGI. Esto se hace mediante campos ocultos en

el *form*, una técnica de implementación que oscurece el trabajo del desarrollador. HyCom permite trabajar directamente con un estado asociado al *form*, manejando internamente el uso de campos ocultos (ahora como una forma de implementación invisible al usuario). Podemos capturar esto con elegancia mediante la clase `HasState`.

```
instance (UIComponent a) => HasState (Form a) FormState where
   getState aForm          = ...
   setState newState aForm = ...
```

El tipo `FormState` básicamente permite encapsular cualquier tipo que pueda convertirse desde y hacia el tipo `String`. Esto es capturado junto a algunas otras operaciones auxiliares por una clase `CGIState`.

```
class (Component s) => CGIState s where
   stateToString   :: s -> String
   ...
```

Si bien los componentes provistos por HyCom permiten manejar programas CGI en forma práctica y elegante, estamos considerando mejorarlo utilizando la biblioteca de CGI definida por John Hughes [Hug99]. La misma utiliza el concepto de *Arrows*, una generalización de las mónadas, definiendo una base teórica sólida para desarrollar programas que pueden "suspenderse"; esta "suspensión" significa que el programa puede detenerse y reanudarse más tarde en el mismo punto en donde se detuvo (en el caso de CGI esto puede verse en los distintos *callbacks* que existen durante una interacción con una aplicación en la WWW).

Templates HTML

A pesar de que HyCom provee un rico conjunto de componentes de interfaz, es posible que el usuario quiera utilizar HTML directamente. Básicamente, el usuario principiante que desea obtener resultados rápidos y está familiarizado con alguna herramienta, preferiría seguir utilizando HTML. Por otro lado, la migración hacia HyCom de *sites* existentes es más rápida y práctica si pueden seguirse utilizando los diseños de interfaz existentes. HyCom soporta estas situaciones mediante *templates* HTML.

Un *template* es un componente de interfaz que puede compilarse a HTML; además, un *template* tiene uno o más *tags* de la forma (`<HyCom ID>`), donde `ID` es el identificador de un componente de interfaz. La función `htmlTemplate` toma el *template* y una lista de componentes de tipo `HTMLComponent` (compilables a HTML) y los inserta en los lugares del *template* marcados por el *tag* (`<HyCom ID>`) correspondiente (a cada componente le corresponde el *tag* parametrizado con su `ID`).

```
htmlTemplate :: (Component template, HTMLCompilable template) =>
                template -> [HTMLComponent] ->
                HTMLTemplate
```

Aclaramos que el tipo `HTMLComponent` es un *wrapper* [GHJV95] sobre cualquier componente compilable en HTML. El tipo utiliza la cuantificación existencial (ver Apéndice A), pudiendo contener a cualquier componente de la clase `HTMLCompilable`.

Para facilitar la comprensión del mecanismo de *templates* consideremos un ejemplo. Supongamos que tenemos un componente `myTemplate`, que al compilarse a HTML resulta en el siguiente segmento de código.

```
<H1> El Siguiente es un Componente Insertado </H1>
<BR>
<HyCom 00001>
```

Supongamos ahora que tenemos otro componente `myText` (cuyo `ID` es `00001`) que al compilarse resulta en el siguiente código HTML.

```
<H2> Este es el Componente </H2>
```

Podemos utilizar el mecanismo de *templates* como se muestra a continuación.

```
templateTest = htmlTemplate myTemplate [myText]
```

El componente `templateTest`, al ser compilado, reemplazará el *tag* (`<HyCom 00001>`) por la versión compilada del componente `myText` (ya que tiene el identificador `00001`). A continuación mostramos el código HTML resultante.

```
<H1> El Siguiente es un Componente Insertado </H1>
<BR>
<H2> Este es el Componente </H2>
```

El mecanismo de *templates* permite gran flexibilidad cuando se desarrollan *web sites*. Además, el mecanismo de *templates* provee una forma muy simple de definir nuevos componentes compilables en HTML.

5.5 Resumen

En este capítulo comentamos la importancia de proveer una forma de integración con plataformas de implementación existentes. Por un lado, la prototipación y generación automática de aplicaciones permiten encarar el diseño mediante prototipos evolutivos, descubriendo errores en forma temprana, y reduciendo costos en la obtención del producto final. Por otro lado, las características específicas de una plataforma pueden ser requeridas por el producto en su versión definitiva.

Vimos cómo la generación automática de aplicaciones y la integración de características específicas es realizada en HyCom. Las clases de tipos permiten capturar el concepto de compilable, brindando un modelo flexible y extensible de compilación de componentes. Vimos finalmente cómo estos conceptos generales se aplican en una biblioteca para la plataforma WWW.

Parte III

Utilización de HyCom

Capítulo 6

Introducción al Desarrollo con HyCom

Los capítulos anteriores mostraron las características básicas del DSEL HyCom. Las diferentes partes del DSEL cubren diversos aspectos de una aplicación, como las consideraciones por la navegación, el diseño de la interfaz del usuario y la interacción con datos persistentes. El desarrollo de una aplicación requiere seguir una serie de pasos sistemáticos, en los cuales se aplican las mencionadas características.

En este capítulo mostramos la forma en que una aplicación se desarrolla utilizando HyCom. Comenzamos esbozando una serie de pasos metodológicos a seguir en el desarrollo mediante HyCom. A continuación presentamos un ejemplo simple de aplicación. El resto del capítulo se focaliza en la resolución del ejemplo utilizando los pasos presentados.

6.1 El Proceso de Desarrollo de Aplicaciones

Desde hace varios años existe un consenso relativamente aceptado sobre la necesidad de adoptar pasos sistemáticos en el desarrollo de hypermedia. Los pasos que deben tomarse han sido motivo de estudio, resultando en diferentes metodologías y enfoques. Sin embargo se está de acuerdo en que el desarrollo pasa por cuatro fases: diseño conceptual, diseño navegacional, diseño de interfaz del usuario e implementación. A lo largo de esta sección analizamos informalmente cada una de estas fases.

6.1.1 Diseño Conceptual

El diseño conceptual consiste en la modelización precisa del dominio de la aplicación. En esta fase se identifican las entidades conceptuales participantes de la aplicación y sus relaciones. Es importante que las entidades sean identificadas en forma independiente a los compromisos de la aplicación a desarrollar posteriormente, con el fin de no opacar detalles importantes en esta primera fase en pos de

necesidades de fases posteriores.

El diseño conceptual puede realizarse mediante diferentes herramientas, como el modelo de entidadades y relaciones (ER) [EN90], el modelo orientado a objetos, etc. El resultado de esta fase es un modelo que refleja el dominio de la aplicación.

6.1.2 Diseño Navegacional

El diseño navegacional consiste en determinar las consideraciones de navegación sobre la información identificada en el modelo conceptual. Dependiendo del enfoque utilizado, los nodos de la hypermedia son derivados de una o más entidades conceptuales. Habitualmente, los *links* se derivan a partir de las relaciones entre entidades existentes en el modelo conceptual.

El modelo navegacional no necesariamente consiste sólo en nodos y *links* derivados directamente de las entidades y relaciones conceptuales. Existen otro tipos de nodos, como nodos índices, que pueden formar parte del esquema conceptual. Además, existen estructuras conceptuales avanzadas, como los contextos navegacionales, que permiten especificar la navegación en un nivel de abstracción mayor que los *links* [SB94].

Existen diferentes tipos de herramientas para el modelado navegacional. Cada metodología de diseño provee un conjunto de herramientas conceptuales para la especificación de los elementos participantes en esta fase.

6.1.3 Diseño de Interfaz del Usuario

La etapa de diseño de interfaz del usuario requiere definir los elementos que el usuario percibirá al utilizar la aplicación. El diseño de la interfaz consiste en determinar qué elementos percibibles (elementos visuales generalmente) se utilizarán para mostrar los elementos del modelo navegacional, y cómo se representará y activará la navegación.

El diseño de interfaz puede tener en cuenta aspectos exclusivos de este nivel, destinados a facilitar el uso de la aplicación. Existe mucho estudio respecto a cómo realizar interfaces del usuario efectivas. Además, existen patrones de diseño de hypermedia [RSG97] que dan guías generales de cómo mejorar la interfaz teniendo en cuenta aspectos especiales de una aplicación hypermedial.

Existen infinidad de formas de especificar el diseño de la interfaz de la aplicación. Por ejemplo, OOHDM sugiere la utilización de ADVs [CL99] para especificar en forma abstracta la apariencia y comportamiento de la interfaz. Ambientes orientados exclusivamente a la WWW proponen la utilización de *templates* HTML que modelizan la apariencia general de un cierto tipo de nodo.

6.1.4 Implementación

Las construcciones de los modelos anteriores son reflejadas en términos de elementos de la plataforma de implementación elegida. Por ejemplo, en la WWW los nodos estarán representados generalmente por páginas, los efectos de interfaz

utilizarán componentes de HTML y probablemente *scripts* mediante JavaScript [KK97] u otro lenguaje de *scripting*. Si se utiliza XML [XML99] en la implementación, la estructura navegacional estará definida probablemente en términos de *tags* específicos del dominio y la interfaz se definirá mediante *style sheets* [CSS99].

6.2 El Proceso de Desarrollo Adoptado en HyCom

HyCom no prescribe la utilización una metodología particular; sin embargo su provisión de componentes para manejar los diferentes aspectos de una aplicación permiten que la adopción de pasos metodológicos sea natural.

En base a los pasos generales que analizamos en la sección anterior esbozamos un ciclo de desarrollo de aplicaciones con HyCom. El ciclo en cuestión cuenta con los pasos de diseño conceptual, diseño navegacional, diseño de interfaz y generación automática de un prototipo. La fase de implementación es reemplazada por la de generación automática, lo cual es posible gracias a las facilidades de prototipación disponibles en HyCom. Aprovechando las ventajas de la prototipación automática adaptamos el ciclo siguiendo una metodología por prototipos evolutivos (o espiral). Ilustramos este ciclo en la Fig. 6.1.

El desarrollo con HyCom se basa en la definición de componentes apropiados para cada una de las etapas de diseño. En primer lugar se modeliza el dominio mediante componentes representando las entidades y relaciones conceptuales. A continuación se definen componentes para los nodos, *links* y contextos navegacionales derivados del modelo conceptual. En la siguiente fase se definen los componentes de interfaz, encargados de la visualización de cada uno de los nodos. Finalmente se refinan detalles de implementación y se genera un prototipo automáticamente. El proceso continúa a través del refinamiento sucesivo de prototipos, hasta obtener la aplicación deseada.

6.3 Una Aplicación Simple

El enfoque de desarrollo mediante HyCom puede verse más claramente a través de su aplicación en un ejemplo simple. A lo largo de esta sección enfocamos el desarrollo de una aplicación sencilla utilizando el ciclo propuesto.

La aplicación a desarrollar es un subconjunto de un sistema de información académico. Básicamente, la misma permite manejar un conjunto de profesores, cursos y carreras, y las relaciones entre los mismos. Por cada profesor se mantienen sus datos personales y los cursos que dicta. Por cada curso se mantiene su nombre, una breve descripción, el profesor que lo dicta y la carrera a la que pèrtenece. Finalmente, por cada carrera se mantiene su nombre, descripción general y la lista de cursos requeridos. Este ejemplo está sobresimplificado para preservar su claridad.

Deseamos desarrollar una aplicación que permita acceso hypermedial a los datos que describimos. Durante el resto de esta sección nos focalizamos en la resolución

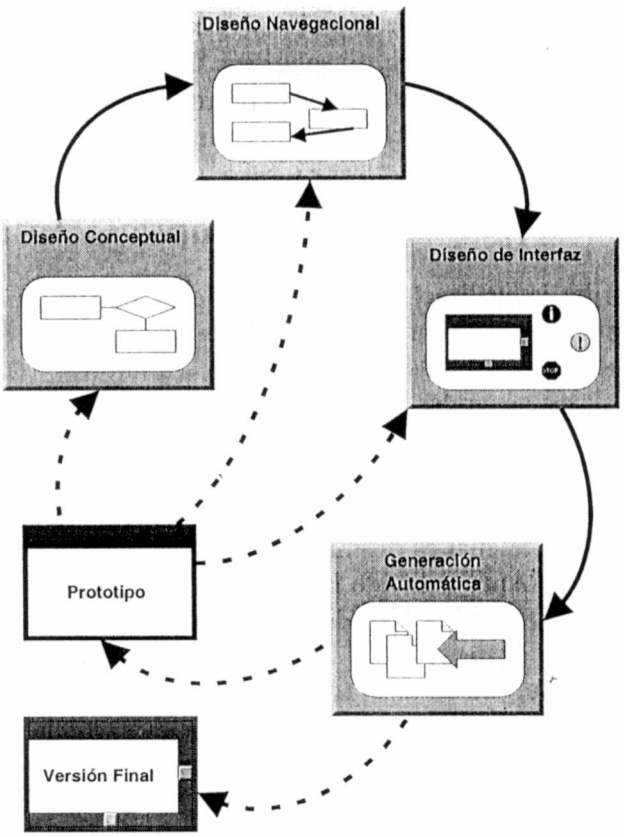

Figura 6.1: Ciclo de Desarrollo Adoptado

de cada uno de los pasos del desarrollo utilizando HyCom.

6.3.1 Diseño Conceptual

Enfocamos el diseño conceptual en dos fases. En la primera fase identificamos claramente el dominio de la aplicación a través de un diagrama de entidadades y relaciones (ER). La segunda fase consiste en la traducción del diagrama ER a un conjunto de componentes.

Los diagramas ER han sido utilizados ampliamente por diversas comunidades para el modelado conceptual. Decidimos utilizarlos en la primera fase del modelado debido a su simplicidad y a la experiencia existente en su utilización (una aplicación más compleja podría beneficiarse de utilizar otro modelo, como el orientado a objetos). El diagrama ER modelando el dominio de nuestra aplicación puede verse en la Fig. 6.2.

La traducción del diagrama ER a un conjunto de componentes nos permite empezar a trabajar en el marco propuesto por HyCom. En un principio parecería

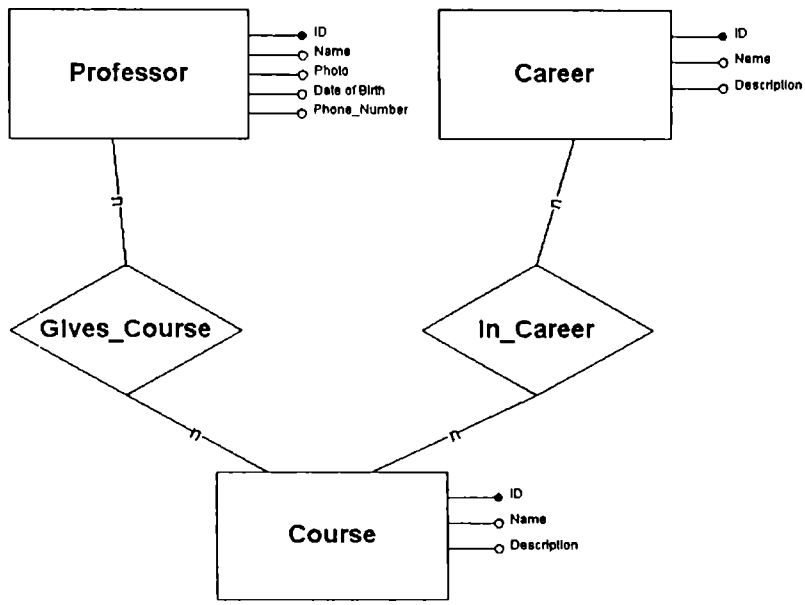

Figura 6.2: Esquema ER Conceptual

innecesario realizar esta traducción en esta primera fase puramente conceptual. Sin embargo expresar el diseño conceptual mediante componentes permite un pasaje más directo al modelo navegacional. Por otro lado, la naturaleza declarativa de HyCom permite expresar conceptos de alto nivel sin sacrificar la claridad.

El proceso de traducción del diagrama ER a un conjunto de componentes es simple y prácticamente inmediato. Por ejemplo, el siguiente es el código HyCom correspondiente a la entidad Professor, que pasa a representarse mediante el componente `Professor`.

```
data Professor = Professor { professorID    :: ID,
                             professorName  :: String,
                             professorPhoto :: String,
                             professorDoB   :: Date,
                             professorPhone :: String,
                           }
```

La definición de un componente se completa realizando la correspondiente instanciación de la clase `Component`.

```
instance Component Professor where
   getID                = professorID
   setID newID professor = professor{professorID = newID}
```

Las relaciones pueden representarse naturalmente mediante funciones. Por ejemplo, la relación entre un profesor y los cursos que dicta puede modelizarse

mediante un tipo de función `CoursesGiven` que dado un profesor retorna la lista de cursos dictados.

```
type CoursesGiven = Professor -> [Course]
```

El tipo `CoursesGiven` se define como un sinónimo de tipos. En un caso real, es más conveniente utilizar un nuevo tipo en vez de un sinónimo, ya que esto permite utilizar más eficientemente el sistema de tipos. Aquí utilizamos un sinónimo para preservar la simplicidad del ejemplo.

La representación del modelo conceptual no necesariamente debe realizarse de la forma mostrada aquí. Podemos representar un modelo ER de otras formas, como por ejemplo, representando una entidad y sus relaciones en un único componente. Análogamente otros modelos conceptuales (como el orientado a objetos) pueden representarse mediante componentes siguiendo estrategias diferentes a la de nuestro ejemplo.

6.3.2 Diseño Navegacional

El modelado navegacional puede concebirse como un filtrado del modelo conceptual, con el agregado de características navegacionales. La primer tarea que enfocamos en el diseño navegacional es la representación de las entidades y las relaciones en términos de nodos y *links*. A continuación definimos los contextos navegacionales que complementan el modelo.

En el enfoque de desarrollo adoptado tomamos la postura de modelizar cada entidad conceptual mediante un nodo y cada relación mediante un *link*. Eventualmente pueden existir también nodos y *links* que no correspondan exactamente a una entidad o relación (como por ejemplo, nodos que modelizan índices generales). Analizaremos este último caso más adelante.

La relación mencionada entre el modelo conceptual y el navegacional puede representarse naturalmente en HyCom mediante funciones. Una función de esta categoría toma uno o más componentes del modelo conceptual, retornando el correspondiente componente navegacional. Podemos ver gráficamente esta relación en la Fig. 6.3.

La aplicación de la política mencionada resulta en un esquema navegacional simple y claro. Tenemos nodos por cada profesor, cada curso y cada carrera. Además tenemos *links* por cada una de las relaciones (por ejemplo, a partir de un curso es posible navegar hacia los profesores que lo dictan). Finalmente decidimos organizar a los profesores, los cursos y las carreras en contextos secuenciales, ordenados por orden alfabético. El esquema se ilustra en la Fig. 6.4.

Los requisitos del modelo navegacional pueden formalizarse directamente mediante HyCom. Comenzamos por definir los componentes correspondientes a cada uno de los nodos. El componente de un nodo mantiene información sobre la correspondiente entidad, y además *anchors* relacionados con los *links* correspondientes.

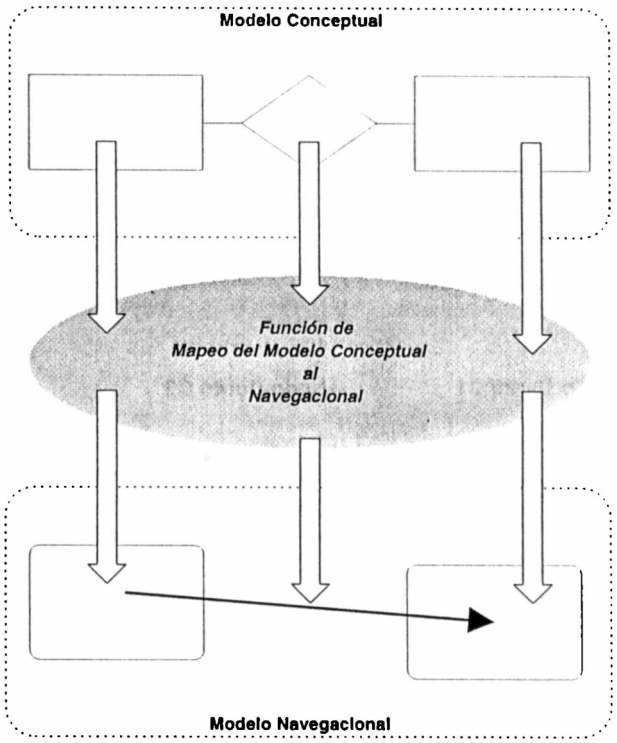

Figura 6.3: Mapeo del Modelo Conceptual al Modelo Navegacional

Mostramos, en este caso, el código correspondiente al nodo de un curso (representamos los *anchors* mediante el tipo `StringBasedAnchor`, que implementa una forma muy simple de *anchor*).

```
data CourseNode = CourseNode
                  { theCourse          :: Course,
                    toCourseCareer     :: StringBasedAnchor,
                    toCourseProfessors :: [StringBasedAnchor],
                  }
```

El tipo `CourseNode` aloja a un componente `Course`, responsable de mantener la información sobre el curso. Además mantiene componentes de tipo `StringBasedAnchor` para modelizar las relaciones de navegación.

La definición del nodo se complementa realizando la instanciación de las correspondientes clases. En primer lugar debemos realizar la instanciación de la clase `Component`, de forma similar a lo que se hizo en el modelo conceptual. Luego realizamos la instanciación de la clase `NodeComponent`, que define las responsabilidades básicas de todo nodo.

```
instance NodeComponent CourseNode where
```

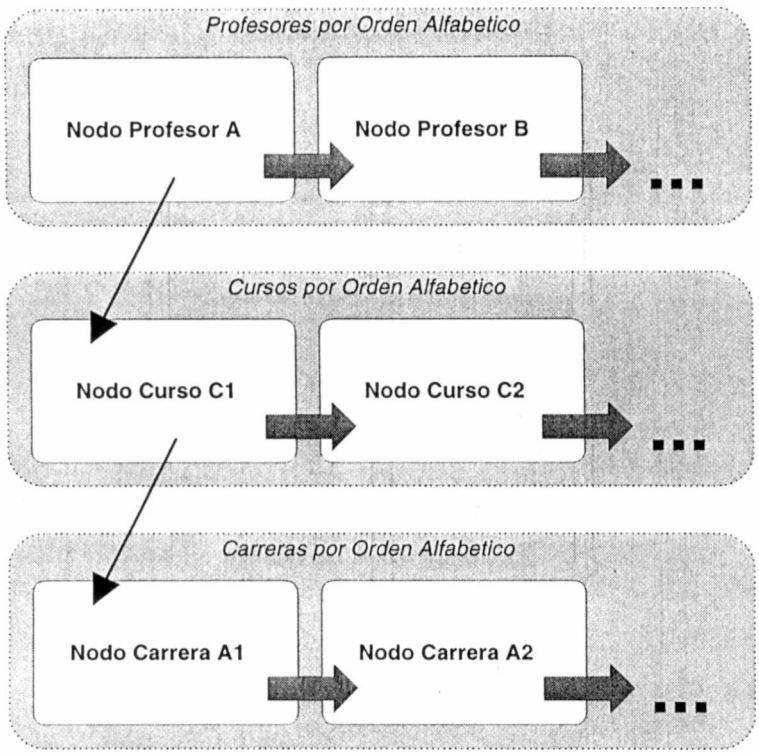

Figura 6.4: Esquema Navegacional Deseado

```
getOutgoingLinks c = Links $
                     map getLink $
                     [toCourseCareer c] ++
                     toCourseProfessors c
```

El operador `$` es simplemente la aplicación de funciones con asociación a derecha y máxima precedencia (de forma que `(f $ g $ h x)` es equivalente a `(f (g (h x)))`). El constructor `Links` hace uso de la cuantificación existencial (ver el Apéndice A) permitiendo que `getOutgoingLinks` pueda retornar cualquier tipo de *link*. La función `getLink` obtiene el *link* a partir del *anchor*. La instanciación define la función `getOutgoingLinks`, que retorna los *links* emergentes de un nodo (en este caso son los que relacionan al curso con su carrera y sus profesores).

La relación entre los componentes conceptuales y los navegacionales se formaliza mediante las funciones de construcción de nodos. Una función de construcción de nodos toma un conjunto de componentes del modelo conceptual, retornando un componente correspondiente a un nodo determinado. De esta forma se tienen funciones apropiadas para cada uno de los nodos de la aplicación.

```
professorNode :: Professor -> ProfessorNode
```

```
professorNode aProfessor
              = ProfessorNode aProfessor (mkAnchors courses)

   where courses = coursesGiven aProfessor
```

La función professorNode toma un componente de tipo Professor y construye un nodo de tipo ProfessorNode. La función coursesGiven representa la correspondiente relación y permite recuperar los cursos dictados por un profesor.

```
coursesGiven :: CoursesGiven
```

El nodo construído por professorNode aloja a un profesor y a un conjunto de *anchors*, construído a partir de los cursos correspondientes al profesor. La función mkAnchors es la encargada de construir los *anchors* a partir de los cursos.

El paso restante en el modelado navegacional consiste en la definición de los contextos. HyCom provee componentes específicos para la construcción de contextos de diferentes tipos. En particular, para contextos secuenciales, HyCom provee la clase SequentialContext que define su funcionalidad. Además provee componentes como SimpleContext que implementan la funcionalidad básica de los contextos secuenciales. Un componente para un contexto como el de los profesores ordenados alfabéticamente puede definirse en forma sencilla a partir de SimpleContext.

```
newtype ProfessorsByName = ProfessorsByName SimpleContext
```

Como en otros casos de definición de componentes, es necesario realizar instanciaciones de las clases apropiadas. Para todo contexto navegacional, además de realizar la instancia de Component, es necesario realizar la instancia de Context.

```
instance Context ProfessorsByName where
   inContextNodeID _ i = transformID "ProfessorsByName" i
   getNodesInContext   = getNodesInContext . getSimpleContext
```

La instanciación especifica que el contexto ProfessorsByName cumple con la funcionalidad básica de todo contexto. Es especialmente interesante ver que la función getNodesInContext puede definirse a partir de la composición de funciones existentes. Este mecanismo de reuso es factible en el paradigma funcional gracias al concepto de función de alto orden. Esta utilización de las funciones de alto orden nos permite reducir la cantidad de código necesario, facilitando la comprensión de las definiciones.

La definición de un contexto secuencial, como el que estamos definiendo, se completa a partir de la instanciación de la clase SequentialContext.

```
instance SequentialContext ProfessorsByName where
   nextInContext     = simpleNext     . getSimpleContext
   previousInContext = simplePrevious . getSimpleContext
```

Un contexto debe tener una función de construcción asociada que permite construírlo a partir de un conjunto de nodos. Para el contexto mostrado la función de construcción es la siguiente.

```
professorsByName :: [Professor] -> ProfessorsByName
professorsByName = ProfessorsByName     .
                   map setProperID
                   sortByProfessorName
```

La función `professorsByName` construye el contexto a partir de los componentes correspondientes a los profesores.

6.3.3 Diseño de Interfaz del Usuario

En esta fase nos focalizamos en la definición de componentes de interfaz, que serán la representación perceptible de los componentes definidos en el modelo navegacional. Definimos la interfaz de un nodo mediante una función que toma el componente correspondiente al nodo en cuestión y retorna el correspondiente componente de interfaz.

Podemos realizar la definición de las interfaces de los nodos mediante la apropiada combinación y transformación de los componentes provistos por HyCom. Los componentes básicos representan elementos típicos de una interfaz gráfica, como una imagen, un campo de texto, etc. Además, HyCom provee varios componentes propios de interfaces hypermediales, como ser los *anchors*. Finalmente, las relaciones de *layout* entre componentes están modelizadas mediante transformadores.

A modo de ejemplo, vemos la especificación de interfaz de un nodo correspondiente a un profesor. En la definición utilizamos componentes básicos como `textLabel` (un campo estático de texto), `image` (una imagen) y `divisionLine` (una línea divisoria). Las relaciones de *layout* se representan mediante los combinadores `/=\` (combinación vertical) y `<=<` (combinación horizontal). Utilizamos además una función `standardUI` que captura el *look-and-feel* general de la interfaz (en este caso, una cabecera y pie de página determinados).

```
professorUI professorNode = standardUI $
                            personalInfo   /=\
                            divisionLine   /=\
                            contextAnchors /=\
                            divisionLine   /=\
                            courses        /=\

   where

      personalInfo   = personalData <=< image photo

      personalData   = (textSize 5 (textLabel name)) /=\
                       textLabel phoneNumber         /=\
```

```
                    textLabel dateOfBirth

courses         = (textSize 4 (textLabel "Courses")) /=\
                  mkAnchorsUI coursesAnchors

contextAnchors = anchorPrev /=\
                  anchorNext
```

En la definición de professorUI omitimos la definición de algunas funciones, como anchorPrev, anchorNext y mkAnchorsUI. Las dos primeras construyen los *anchors* que activan la navegación a los nodos previo y siguiente en el contexto, respectivamente; la función mkAnchorsUI toma una lista de *anchors* de tipo StringBasedAnchor, retornando una representación visual consistente en la disposición vertical de los *anchors*.

```
mkAnchorsUI = verticalBox . map mkAnchorUI
```

La función verticalBox toma una lista de componentes y retorna un componente resultante de apilarlos visualmente. La función (map mkAnchorUI) aplica mkAnchorUI a la lista de *anchors*, siendo mkAnchorUI la función de visualización de un *anchor*.

```
mkAnchorUI aStringBasedAnchor = anchoredUI link text

   where link = getLink aStringBasedAnchor
         text = textLabel (getString s)
```

La función standardUI merece una mención especial. La misma permite capturar el *look-and-feel* de los nodos de la aplicación, consistente en una cabecera y un pie de página estándares. Este ejemplo simple muestra cómo una estructura visual común puede capturarse fácilmente en HyCom a través de funciones.

```
standardUI aUI = standardHeader /=\
                 divisionLine   /=\
                 aUI            /=\
                 divisionLine   /=\
                 standardFooter

   where standardHeader = image "Logo.Gif"
         standardFooter = image "Footer.Gif"
```

La definición de interfaces del usuario para el resto de los nodos sigue los principios generales que vimos en esta sección. Como podemos ver, la especificación de interfaz resultante es declarativa e independiente de plataforma. Además, podemos utilizar funciones para capturar el *look-and-feel* de los nodos. Finalmente, las funciones de alto orden nos permiten reutilizar funciones, ahorrando código y ganando claridad en las definiciones.

6.3.4 Generación de un Prototipo

Una vez definidos los aspectos básicos de la aplicación, podemos proceder a la generación automática de un prototipo. El prototipo generado permite evaluar la aplicación, refinando los aspectos insatisfactorios en forma sucesiva, hasta cubrir completamente los requerimientos.

Podemos generar un prototipo en forma automática mediante una función de compilación para la plataforma adecuada. Previamente a la generación del prototipo necesitamos resolver ciertos detalles implementativos, como la carga de los datos y la generación de la especificación compilable a partir de los mismos.

En general, los datos de la aplicación se encuentran almacenados en una base de datos o en un conjunto de archivos. La relación entre las entidades y el medio de almacenamiento puede encapsularse mediante un *retriever*. La recuperación de los datos se realiza mediante las operaciones provistas por el *retriever* con lo cual la aplicación puede abstraerse de los detalles de la conexión con el almacenamiento. Suponiendo que tenemos un *retriever* `aRetriever` para los datos de nuestra aplicación, el fragmento de código monádico (ver Apéndice A) muestra cómo pueden recuperarse los conjuntos de entidades.

```
do
   ...
   professors <- retrieve aRetriever :: IO [Professor]
   courses    <- retrieve aRetriever :: IO [Course]
   careers    <- retrieve aRetriever :: IO [Career]
   ...
```

Las signaturas de tipo explícitas permiten al compilador determinar que instancia de `retrieve` debe utilizar en cada caso; esto sucede puesto que `retrieve` es una función sobrecargada cuya instancia particular queda determinada por la signatura de tipo.

La generación de una especificación compilable consiste en generar nodos a partir de los datos recuperados y luego generar las correspondientes interfaces. Una especificación compilable consiste de una serie de componentes que se compilan juntos (por ejemplo, las interfaces de los nodos de una aplicación). En el siguiente ejemplo mostramos cómo a partir de un conjunto de componentes recuperados de una base de datos podemos construir una especificación; la función `simpleSpecification` toma los conjuntos de entidades y relaciones, las funciones de construcción de nodos e interfaces y retorna una especificación.

```
do
   ...
   professors <- retrieve aRetriever :: IO [Professor]
   courses    <- retrieve aRetriever :: IO [Course]
   ...
   spec       <- simpleSpecification
                       professors    courses    careers
```

```
coursesGiven  inCareer
professorNode courseNode careerNode
professorUI   courseUI   careerUI
```

La expresión mostrada construye los nodos a partir de las entidades y relaciones, aplicando las funciones de construcción adecuadas. De la misma manera construye las interfaces a partir de los nodos. El resultado es una especificación que puede ser tomada por una función de compilación apropiada, generando una aplicación. El siguiente código ejemplifica este proceso con el uso de la función de compilación a HTML, donde `compilationInfo` es la información de compilación.

```
hyComToHTML compilationInfo spec
   where compilationInfo = generateCompilationInfo spec
```

La función `hyComToHTML` toma la especificación junto con cierta información de compilación, y genera un conjunto de páginas HTML correspondientes al prototipo. La información de compilación tiene que ver con los nombres de las páginas generadas y detalles relacionados. Esta información puede ser provista por el usuario o generada automáticamente, como hicimos aquí mediante la función `generateCompilationInfo`.

Podemos ver una página HTML correspondiente al prototipo generado en la Fig. 6.5. Hemos marcado en la misma las correspondencias con la especificación de interfaz que realizamos en la sección anterior, para facilitar la comparación.

6.3.5 Refinamiento del Prototipo

La etapa siguiente a la generación de un prototipo consiste en su evaluación y eventual refinamiento. Una vez obtenida la primera versión de la aplicación analizamos si la misma cumple con los requisitos deseados. En caso de que el prototipo muestre deficiencias en alguna fase del modelado, volvemos a esa fase, la refinamos y generamos el prototipo nuevamente.

Supongamos que deseamos refinar el esquema navegacional de forma de disponer un índice para cada contexto. HyCom permite realizar esto mediante los contextos indexados. Podemos extender los contextos definidos previamente para que cumplan los requisitos necesarios para ser indexados. Realizar esto implica hacer los contextos instancias de la clase `IndexedContext`, de forma similar a lo que hicimos previamente para otras clases.

```
instance IndexedContext ProfessorsByName String where
   contextIndex = simpleIndex . getSimpleContext
```

Deseamos cambiar también la interfaz para visualizar los índices de los contextos. A continuación vemos el código refinado de una interfaz para un nodo correspondiente a un profesor.

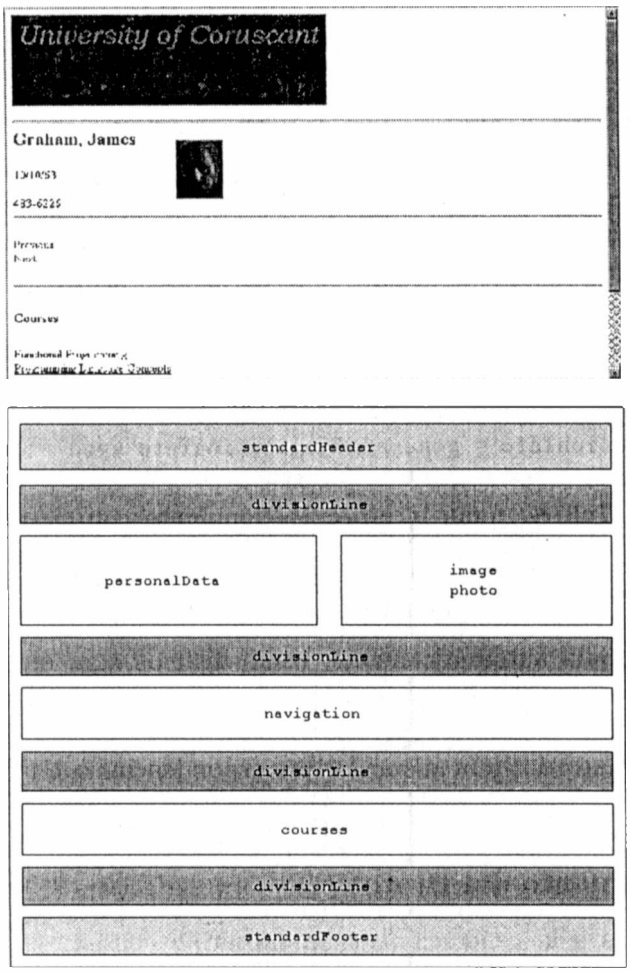

Figura 6.5: Primer Prototipo

```
professorUI professorNode = standardUI2 $
                            personalInfo   /=\
                            contextAnchors /=\
                            divisionLine   /=\
                            contextIndex   /=\
                            divisionLine   /=\
                            courses
```

En la definición de `professorUI` usamos las funciones `personalInfo` (información personal del profesor), `contextAnchors` (*anchors* a los nodos siguiente y anterior en el contexto), `contextIndex` (índice del contexto) y `standardUI2` (versión refinada de `standardUI`). Debido a que las definiciones de las mismas no aportan nuevos

conceptos, no las mostramos.

Ahora podemos generar un nuevo prototipo. El mismo se muestra en la Fig. 6.6.

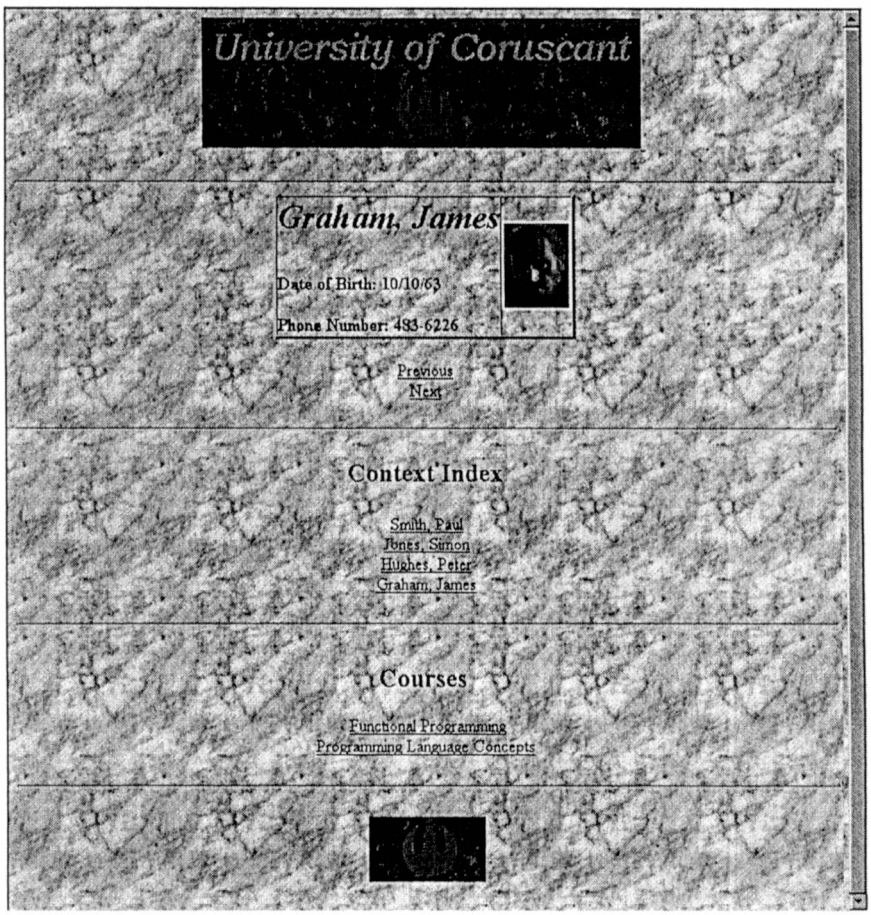

Figura 6.6: Segundo Prototipo

El proceso continúa en forma iterativa hasta obtener la versión final. Es importante destacar que las definiciones de los componentes conceptuales, navegacionales y de interfaz son especificaciones y a la vez implementaciones de la aplicación. Esto es producto de la naturaleza declarativa de HyCom, que reduce la distancia entre especificación e implementación.

6.4 Resumen

En este capítulo mostramos la forma en que una aplicación se desarrolla utilizando HyCom. Presentamos un ciclo de desarrollo simplificado, basado en las fases comúnmente aceptadas en la ingeniería de aplicaciones hypermediales. En base al ciclo desarrollamos un ejemplo simple.

Capítulo 7

Bibliotecas de Autoría

El capítulo anterior presentó la forma general del desarrollo de una aplicación mediante HyCom. Si bien el diseño de una aplicación puede realizarse de la forma vista, es posible optimizar el proceso mediante el uso de una biblioteca de autoría. Una biblioteca de autoría surge de la utilización de la metodología DSEL al nivel de una aplicación, permitiendo el reuso de estructuras y experiencia en el desarrollo de aplicaciones hypermediales con dominios conceptuales similares.

Comenzamos mostrando cómo la metodología DSEL puede aplicarse nuevamente, permitiendo reuso de elementos de diseño para reducir el esfuerzo de desarrollo. A continuación explicamos los pasos involucrados en el diseño de una biblioteca de autoría y mostramos un ejemplo de la aplicación de la metodología DSEL.

7.1 Aplicación de la Metodología DSEL a Nivel de Aplicación

El desarrollo de una aplicación hypermedial sigue típicamente pasos similares a los vistos en el ejemplo del Cap. 6. Resulta claro que para aplicaciones con dominios conceptuales similares o idénticos, las decisiones de diseño se repetirán en forma igual o parecida una y otra vez.

De la misma forma en que HyCom captura los conceptos de diseño generales de las aplicaciones hypermediales, es posible llevar la metodología DSEL un paso más adelante y capturar los elementos de diseño de dominios conceptuales más específicos. De esta forma se reutilizarían las decisiones de diseño a diferentes niveles y el desarrollador vería aliviada su tarea.

Podemos llevar la metodología DSEL a este nivel, embebiendo en HyCom DSELs para dominios de aplicaciones hypermediales particulares. Esto nos lleva al concepto de biblioteca de autoría. Una *biblioteca de autoría* es un DSEL para un tipo particular de aplicación (por ejemplo, un sistema de información académico) y consiste básicamente en los siguientes puntos: un modelo conceptual general del dominio elegido, un modelo navegacional general definido sobre el modelo conceptual, diferentes interfaces de usuario extensibles y herramientas diversas

(persistencia, herramientas específicas al dominio, etc.).

Los modelos conforman la biblioteca de autoría y están capturados por código HyCom genérico y extensible. De esta forma el desarrollador puede utilizar el modelo provisto, o extenderlo para cubrir los requerimientos específicos de su aplicación. Además, las bibliotecas proveen las definiciones para manejar persistencia, diferentes interfaces de usuario, etc. Claramente, es necesario un cuidadoso diseño para permitir no sólo tener un modelo suficientemente general y extensible, sino una implementación que permita el uso práctico y la extensibilidad sin necesidad de modificar la biblioteca base.

La idea detrás de la biblioteca de autoría consiste no sólo en reutilizar componentes comunes a un dominio, sino también decisiones de diseño y arquitecturas. La decisión de adoptar una biblioteca de autoría en el desarrollo implica que el diseñador deberá aceptar ciertas pautas de diseño y estructura establecidas por la misma. A cambio de esto, el desarrollador verá aliviada su tarea al reutilizar componentes, arquitecturas y elementos de diseño existentes.

7.2 Diseño de una Biblioteca de Autoría

El desarrollo de una biblioteca de autoría requiere de un cuidadoso diseño. En líneas generales, el diseño de una biblioteca de autoría sigue los mismos pasos que el diseño de una aplicación, pero debe tener en cuenta la generalidad y las posibilidades de extensión.

El desarrollo de los modelos conceptual y navegacional de una biblioteca se realiza de la misma forma que para una aplicación. Sin embargo, deseamos que la representación de los modelos mediante componentes sea suficientemente flexible como para no prescribir una implementación particular (mencionamos en el Cap. 6 que existen diferentes formas de representar un modelo utilizando HyCom). Por esta razón, en vez de definir componentes concretos representando los elementos del modelo (entidades, relaciones, nodos, etc.), utilizamos clases de tipos para describir sus responsabilidades; además, proveemos implementaciones por defecto de componentes que cumplen con esas responsabilidades. El usuario final puede elegir utilizar las implementaciones provistas o definir sus propias implementaciones (posiblemente, utilizando las provistas como base) e instanciarlas para las clases adecuadas.

La definición de las interfaces de usuario debe tener en cuenta también las consideraciones por la extensibilidad. En particular, es conveniente que una interfaz para un nodo particular no asuma sobre el mismo más responsabilidades que las descritas por su clase de tipos; esta consideración permite que una misma interfaz pueda seguir siendo utilizada a pesar de un cambio en la implementación de los componentes.

En general, una biblioteca de autoría cuenta también con un conjunto de herramientas específicas del dominio modelado. Areas potenciales para el desarrollo de herramientas son la validación de las estructuras de navegación, la validación

de los datos, el manejo de datos almacenados, etc.

Finalmente, debido a que una biblioteca de autoría está basada en el reuso de experiencia en cierto dominio, el refinamiento de la misma es una actividad natural en todo su ciclo de vida. Normalmente, el desarrollo de aplicaciones mediante la biblioteca aportará importantes guías sobre los puntos débiles de la misma y sobre las potenciales extensiones. El refinamiento sucesivo es, por lo tanto, un punto fundamental en el éxito de una biblioteca de autoría.

7.3 Biblioteca de Autoría de Sites Académicos

Los sistemas de información académicos son un ejemplo de aplicación típico en donde la estructura conceptual y navegacional es generalmente muy similar entre aplicaciones. Debido a la amplia proliferación y variedad de *sites* implementando sistemas académicos, consideramos que una biblioteca de autoría para este dominio resulta un caso de estudio muy interesante.

En base al análisis de varios *sites* existentes desarrollamos una biblioteca de autoría para *sites* académicos. El desarrollo de esta biblioteca sigue el ciclo básico que explicamos anteriormente. A lo largo de esta sección explicamos cómo llevamos a cabo cada una de las fases de la definición de la biblioteca de autoría.

7.3.1 Modelo Conceptual

Comenzamos el modelado conceptual realizando un análisis de los requisitos comúnmente encontrados en este tipo de *sites*. Uno de los puntos atractivos de elegir un *site* académico como primer caso de estudio es la amplia variedad de *sites* de este tipo. Luego del análisis acordamos los requerimientos generales de un *site* académico. En general decidimos que la estructura adoptada debe ser capaz de manejar profesores, alumnos, carreras, cursos, departamentos, investigadores, laboratorios y publicaciones.

Una vez establecidos informalmente los requisitos, nos focalizamos en la formalización de los mismos mediante un diagrama ER. Decidimos utilizar el diagrama ER debido a que lo encontramos suficientemente expresivo para el dominio en cuestión (dominios más complejos pueden beneficiarse del modelado orientado a objetos). Mostramos una parte del diagrama en cuestión en la Fig. 7.1. Decidimos no mostrar el diagrama completo a fin de focalizarnos exclusivamente en los aspectos que desarrollamos en detalle a lo largo del capítulo.

Habiendo desarrollado el esquema ER, el siguiente paso consiste en expresarlo mediante componentes. Como mencionamos previamente, no definimos componentes concretos para representar las entidades conceptuales, sino que utilizamos clases de tipos para preservar la generalidad. Las clases definen el comportamiento específico que deben cumplir los componentes que representen a cada una de las entidades. Además, utilizando el concepto de clase podemos representar fácilmente las relaciones de especialización y generalización.

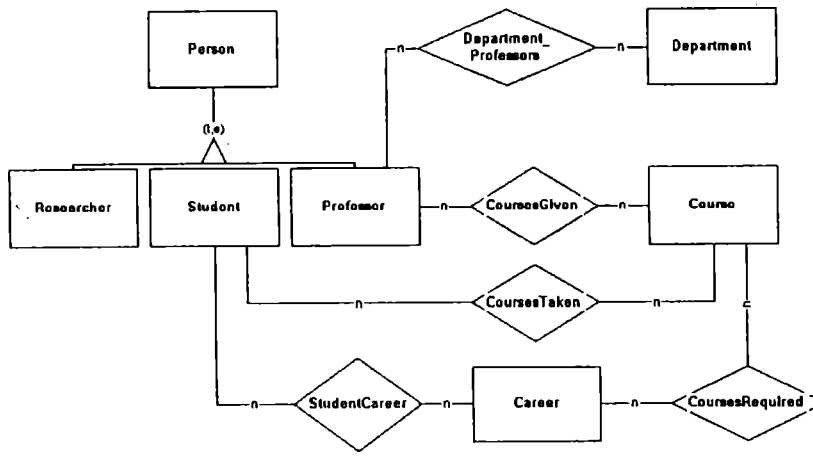

Figura 7.1: Diagrama ER de un *Site* Académico

A modo de ejemplo vemos cómo representar la parte de la jerarquía de personas. En primer lugar definimos una clase `Person` que explicita las operaciones requeridas para todo componente que represente una entidad Person.

```
class (Component p) => Person p where
   getPersonName  :: p -> String
   getDateOfBirth :: p -> String
   getAddress     :: p -> String
   getPhoneNumber :: p -> String
   getPersonPhoto :: p -> String
```

La definición nos dice que `Person` es una subclase de `Component`, heredando las responsabilidades de todo componente. Además, un componente que sea instancia de la clase debe proveer operaciones para acceder a ciertos atributos; estos atributos se corresponden con los atributos de la entidad conceptual Person (por ejemplo, la operación `getPersonName` permite recuperar el nombre de la persona).

El siguiente paso en la modelización de la jerarquía consiste en definir las subclases correspondientes a las entidades Professor, Student y Researcher. Para el caso particular de Professor debemos heredar las responsabilidades de la clase `Person` y agregar operaciones para las responsabilidades específicas de un profesor (conocer los departamentos en los que trabaja, y los cursos que dicta).

```
class (Person p) => Professor p where
   getCoursesGiven :: p -> [ID]
   getDepartments  :: p -> [ID]
```

La operación `getCoursesGiven` permite obtener los `IDs` de los componentes correspondientes a los cursos que dicta el profesor. Análogamente, la operación

`getDepartments` permite recuperar los `IDs` de los componentes correspondientes a los departamentos.

El proceso de representación mediante clases sigue en forma análoga al ejemplo mostrado. La estructura de clases correspondiente al diagrama ER visto se ilustra en la Fig. 7.2. Debemos aclarar que las declaraciones tienen un estilo similar a un registro debido a la naturaleza particular de los datos modelados.

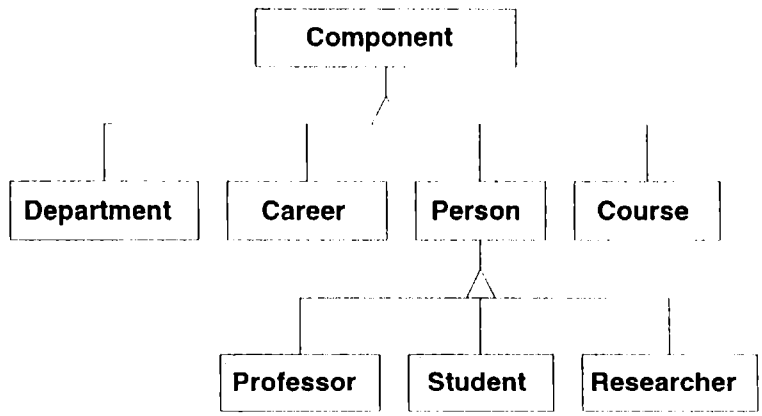

Figura 7.2: Diagrama de Clases de Tipos de un *Site* Académico

La jerarquía de clases resultante especifica la estructura general de los componentes que modelizan el dominio. En el desarrollo de una aplicación se utilizan componentes concretos que implementan las operaciones especificadas por la jerarquía. HyCom provee implementaciones por defecto, sobre las cuales el desarrollador puede basar las propias. Por ejemplo, el tipo `Person` implementa la funcionalidad básica de la clase homónima.

```
data Person = Person { personID      :: ID,
                       personName    :: String,
                       dateOfBirth   :: String,
                       personAddress :: String,
                       personPhone   :: String,
                       personPhoto   :: String
                     }
```

Para completar la definición del componente en cuestión, es necesario realizar las instanciaciones de las clases correspondientes. Por ejemplo, la instanciación de la clase `Person` se muestra a continuación.

```
instance Person Person where
   getPersonName  = personName
   getDateOfBirth = dateOfBirth
   getAddress     = personAddress
   getPhoneNumber = personPhone
   getPersonPhoto = personPhoto
```

Los tipos provistos por defecto implementan la funcionalidad de la clase en forma trivial (como en el ejemplo mostrado). Es usual que el usuario desee definir sus propios componentes concretos, probablemente por motivos de eficiencia en la representación (por ejemplo, es viable definir los componentes directamente como tablas relacionales). Los tipos por defecto, sin embargo, permiten ahorrar esfuerzo durante las etapas de prototipación inicial.

7.3.2 Modelo Navegacional

Comenzamos la definición del modelo navegacional estableciendo las relaciones entre el modelo conceptual y los requerimientos de navegación. En general, la estructura de navegación puede inferirse a partir de la estructura del modelo conceptual siguiendo un conjunto de reglas sistemáticas. Métodos como OOHDM y RMM especifican reglas de este tipo. Para el ejemplo que estamos desarrollando utilizamos reglas basadas en las que esbozamos en el Cap. 6 y que son globalmente similares a las especificadas por los métodos mencionados.

Comenzamos la definición del modelo navegacional definiendo los nodos y los *links*. En general hacemos corresponder un nodo a cada entidad del modelo conceptual y un *link* a cada relación. Una excepción posible a esta regla la constituyen las jerarquías con cobertura total y exclusiva, en donde representamos con nodos sólo las hojas de la jerarquía pero no la raíz. Otros tipos de excepciones (como por ejemplo, nodos que no se corresponden con entidades conceptuales) corresponden generalmente a requerimientos puntuales de la aplicación final. Un ejemplo típico está dado por un nodo de presentación de la aplicación. Debido a que este último tipo de nodo responde a necesidades muy particulares, no lo consideramos en el desarrollo de la biblioteca (ejemplos de este tipo de nodos pueden verse en el Cap. 8).

A modo de ejemplo vemos la representación de un nodo correspondiente a la entidad Professor. Utilizamos clases para definir las responsabilidades de los componentes en cuestión, por las mismas razones que esbozamos al desarrollar el modelo conceptual.

```
class (NodeComponent p, Professor p) => ProfessorNode p where
   toCoursesGiven :: p -> [StringBasedAnchor]
   toDepartments  :: p -> [StringBasedAnchor]
```

La clase `ProfessorNode` es subclase de `NodeComponent` y `Professor`. La operación `toCoursesGiven` permite obtener una lista de *anchors* con *links* a los nodos de los cursos dictados por el profesor. De forma similar, la operación `toDepartments` permite obtener *anchors* asociados con los nodos de los departamentos en los que trabaja el profesor. El tipo `StringBasedAnchor` es un tipo básico provisto por HyCom, y modeliza una forma muy simple de *anchor*.

En general, el proceso de definición de las clases navegacionales es similar al adoptado para las clases conceptuales. La principal diferencia está dada por las

relaciones, que ahora se transforman en *anchors* asociados a los correspondientes *links*. El esquema de clases resultante se muestra en la Fig. 7.3.

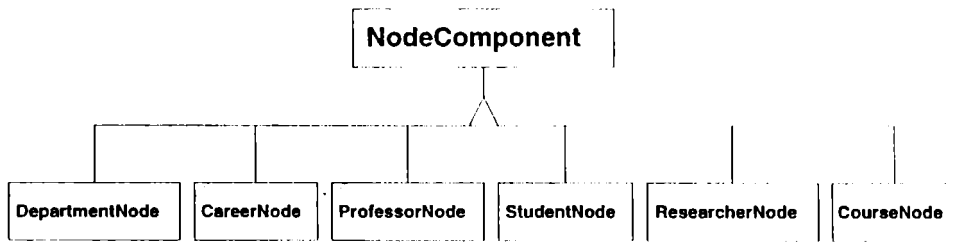

Figura 7.3: Diagrama de Clases de Tipos del Modelo Navegacional

La siguiente fase del modelado navegacional consiste en la definición de contextos de navegación. Los contextos, en general, también pueden inferirse de acuerdo a ciertas reglas (por ejemplo, la estructura de nodos y *links* definida en el paso anterior). El criterio que adoptamos para el ejemplo tiene algunas similitudes al sugerido por OOHDM.

En general determinamos que por cada relación de *linking* con cardinalidad 1 a n, existe un contexto secuencial. Una relación de este tipo está dada cuando desde un nodo determinado podemos acceder a n nodos relacionados con el mismo. Por ejemplo, desde el nodo de un profesor podemos acceder a todos los nodos de sus cursos.

Este esquema básico de contextos puede verse fácilmente con el caso de un profesor y sus cursos. Si estamos accediendo al nodo de un profesor (*Profesor A*) y navegamos hacia uno de sus cursos (*Curso D*), nos encontraremos en el nodo del curso y en el contexto de *todos los cursos dictados por el Profesor A*. Si en el nodo del curso en cuestión decidimos navegar hacia el siguiente curso, navegaremos a otro curso del mismo profesor (o sea, a otro curso en el mismo contexto). Esta estructura se ilustra en la Fig. 7.4.

La regla de definición de contextos describe la forma de generar ciertos contextos, pero no prohíbe la existencia de otros contextos que no se ajusten a la misma. En particular, las formas de *linking* que no sean de la cardinalidad especificada no se atienen a la regla. Para el caso de la biblioteca que estamos desarrollando incluímos también contextos para acceder a todos los profesores, departamentos, carreras y cursos por orden alfabético. Los mencionados casos se corresponden con relaciones de *linking* que no son cubiertas por la regla.

Los contextos se definen mediante componentes concretos. HyCom provee varios componentes y clases que permiten el modelado de contextos, como se explica en el Cap. 4. El siguiente es el código del tipo que implementa el contexto de *todos los cursos dictados por un profesor*.

```
data CoursesGivenByProfessor = CoursesGivenByProfessor SimpleContext
                                                       String
```

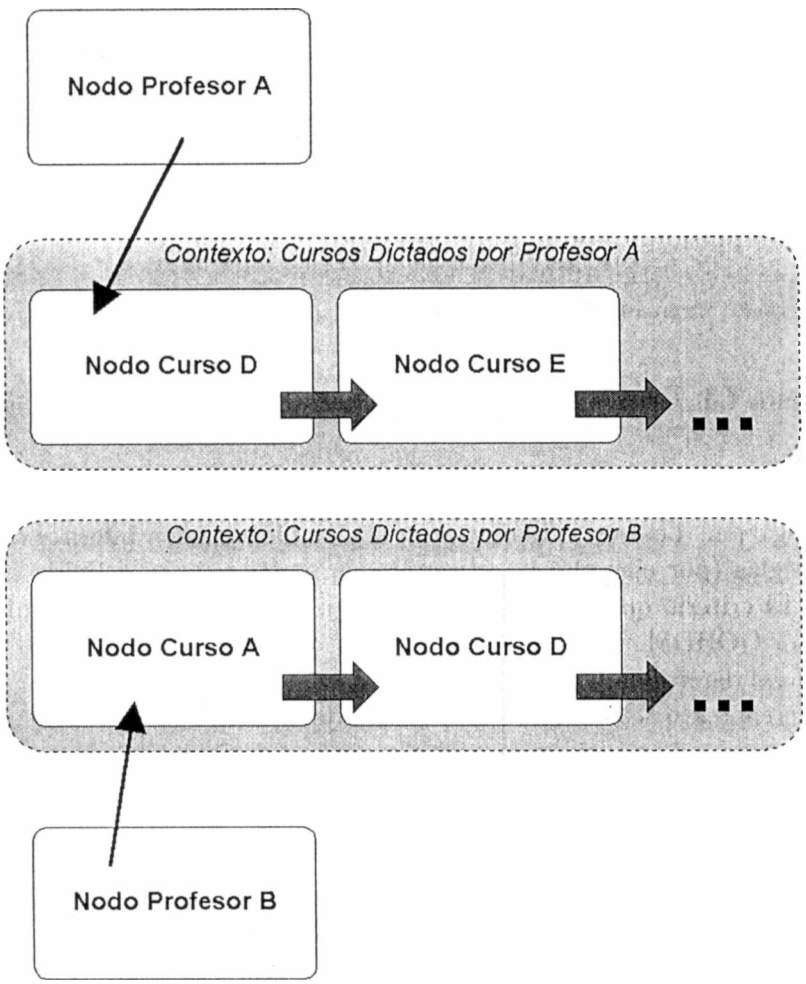

Figura 7.4: Esquema de los Contextos Requeridos

El tipo `CoursesGivenByProfessor` aloja un `SimpleContext` y un `String`. El tipo `SimpleContext` es un tipo provisto por HyCom que provee la funcionalidad básica de un contexto. Incluímos el `String` para mantener el nombre del profesor al que corresponde el contexto; esto es utilizado por algunas de las operaciones para manejar el componente, como por ejemplo, la de obtener la descripción del contexto.

La definición del componente para el contexto continúa mediante la instanciación de las clases correspondientes. Las mismas se hacen de forma análoga al ejemplo visto en el Cap. 6, por lo cual las omitimos.

Para finalizar con la etapa de modelado navegacional, se definen componentes por defecto, implementando la funcionalidad básica de las clases (esto corresponde a las mismas razones que mencionamos para el modelo conceptual). El siguiente es el código correspondiente al componente `ProfessorNode`.

```
data ProfessorNode = ProfessorNode
                     { theProfessor   :: Professor,
                       toCoursesGiven :: [StringBasedAnchor],
                       toDepartments  :: [StringBasedAnchor],
                       currentContext :: ASimpleContext
                     }
```

Esta definición permite mantener datos sobre el profesor, sobre los *links* del nodo y sobre el contexto en el cual se encuentra el nodo. El tipo `ASimpleContext` utiliza la cuantificación existencial (ver Apéndice A) para alojar cualquier tipo de contexto (esto incrementa la flexibilidad de la implementación).

El componente se termina de definir mediante la instanciación de las correspondientes clases. La instanciación sigue los lineamientos generales que vimos para otras instanciaciones, por lo cual la omitimos.

7.3.3 Diseño de Interfaz del Usuario

La interfaz del usuario constituye generalmente el elemento que más cambia de una aplicación a la otra. De hecho, la interfaz de usuario de una aplicación tenderá a evolucionar por factores de usabilidad, cuestiones estéticas, etc. Es de esperar entonces que diferentes aplicaciones desarrolladas con la biblioteca tengan requerimientos de interfaz muy dispares. Por estas razones, es difícil proveer definiciones que satisfagan completamente las necesidades de aplicaciones diferentes.

Considerando los puntos mencionados adoptamos la política de proveer componentes básicos de interfaz que sirvan a modo de prototipos. Los mismos pueden ser utilizados en las fases iniciales del prototipación, y también como base para el desarrollo de nuevos componentes.

Una segunda consideración en el desarrollo de los componentes básicos es diseñar los mismos de forma de proveer el mayor grado de reusabilidad posible. La decisión de diseño tomada para proveer esto fue la de separar la presentación de los datos puros (es decir, aquellos que provienen de las entidades conceptuales), de los datos con implicaciones navegacionales (como por ejemplo, los *anchors*).

Para facilitar la comprensión de las implicaciones de nuestra decisión de diseño mostramos un ejemplo. La siguiente es la definición de una función que asocia una interfaz del usuario a un componente correspondiente a una entidad conceptual (debido a que su signatura de tipo es complicada, nos limitamos a comentar sólo la parte más significativa de la misma).

```
-- professorUI :: (HTMLCompilable t, Professor p) => ...

professorUI template professor = setID (getID professor) $
                                 makeTemp template

  where makeTemp t = htmlTemplate t [nameC,
                                     photoC,
                                     addressC,
```

```
                                    phoneC,
                                    dateC]
nameC       = HTMLComponent $
              setID (iD "InsertedName") $
              textLabel $ getPersonName professor
```

Esta función utiliza el mecanismo de *templates* HTML (ver el Cap. 5) para proveer una interfaz para componentes que sean instancia de la clase `Professor`. El constructor `HTMLComponent` utiliza cuantificación existencial (ver Apéndice A) para encapsular componentes HTML de diferentes tipos. Las definiciones de los otros componentes insertados en el *template* no se muestran, al ser muy similares a la de `nameC`.

El componente definido por `professorUI` permite visualizar los datos de una entidad, sin ningún tipo de consideración por los aspectos navegacionales. Estos últimos son tenidos en cuenta por la definición de interfaz para el correspondiente nodo. La misma toma una función `mkProfessorUI` (que puede ser, por ejemplo, `professorUI`) y un componente `professorNode`, retornando una definición de interfaz para el componente. La función retorna una definición de interfaz en la que sólo se tienen en cuenta los aspectos navegacionales (los contextos son visualizados por la función `inContextUI` y los *anchors* por `mkAnchorsUI`). La visualización de datos puros es delegada a la función `mkProfessorUI` tomada como parámetro, lo cual es posible gracias a las funciones de alto orden.

```
professorNodeUI mkProfessorUI
                professorNode
                = setID (getID professorNode) $
                  inContextUI professorNode $
                  (mkAnchorsUI (iD "InsertedCourses")
                               toCoursesGiven professorNode)  /=\
                  (mkAnchorsUI (iD "InsertedDepartments")
                               toDepartments professorNode)   /=\
                  mkProfessorUI professorNode
```

La estructura mostrada por el ejemplo permite gran flexibilidad, debido a que es posible combinar diferentes funciones de visualización de entidades (como `professorUI`) con diferentes funciones de visualización de nodos (como por ejemplo `professorNodeUI`). De esta forma, el usuario puede elegir cambiar parte de la interfaz del usuario (por ejemplo, los aspectos navegacionales) sin necesidad de cambiar el resto. El siguiente código muestra un ejemplo de varias interfaces definidas a partir de distintas combinaciones.

```
interface1 = professorNodeUI  professorUI
interface2 = professorNodeUI2 professorUI
interface3 = professorNodeUI  professorUI2
...
```

La facilidad de combinación se debe a una decisión de diseño clave. La misma consiste en que las interfaces básicas que definimos se basan exclusivamente en la estructura de las clases, y no en tipos concretos. Esto significa que, por ejemplo, la función `professorUI` otorga una visualización a cualquier componente que sea instancia de la clase `Professor` sin importar su tipo. Análogamente, la función `professorNodeUI` permite visualizar cualquier componente de la clase `ProfessorNode`. Las funciones de visualización extraen los datos del componente utilizando operaciones de la clase y no operaciones del tipo, lográndose así la generalidad deseada.

Las interfaces para el resto de los componentes se definen de forma similar. En cada caso proveemos, además, diferentes opciones (por ejemplo, versiones utilizando *templates* y versiones utilizando combinadores). A modo de ejemplo, diferentes combinaciones de componentes de interfaz pueden verse en las Figs. 7.5 y 7.6.

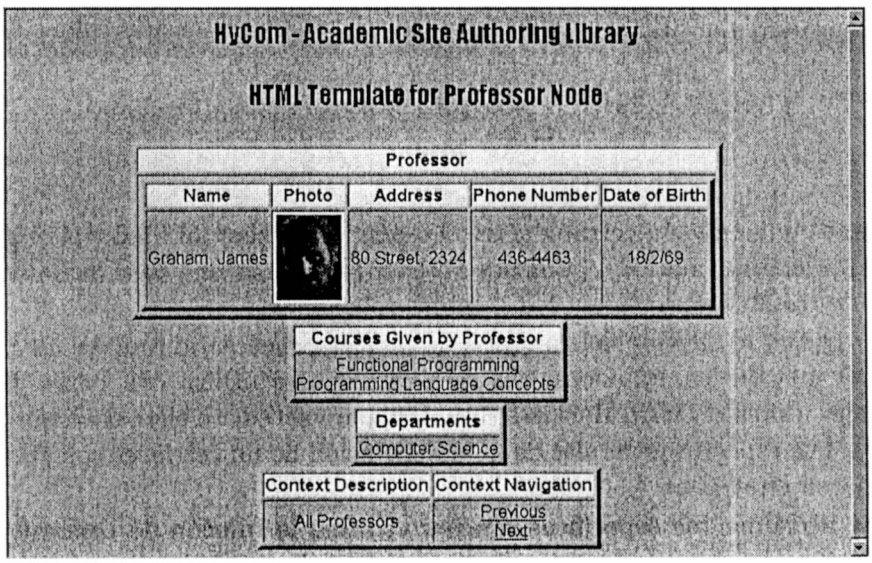

Figura 7.5: Interfaz del Usuario para el Nodo Professor

7.3.4 Definición de Herramientas Específicas

Los grupos más importantes de herramientas son las concernientes a los aspectos de persistencia y generación de especificaciones. Otros grupos potenciales se ocupan de la validación de especificaciones o proveen herramientas muy específicas al dominio de la aplicación. Para nuestro ejemplo nos focalizamos en el primer grupo de herramientas.

La persistencia se maneja comúnmente mediante bases de datos relacionales. El almacenamiento en una base de datos relacional implica en primer lugar la representación del esquema conceptual con un conjunto de tablas. Debido a que

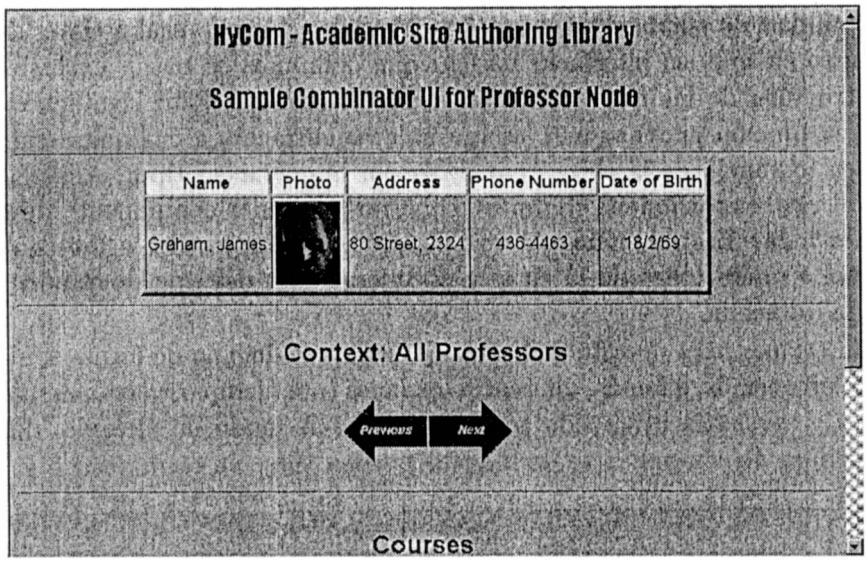

Figura 7.6: Interfaz Alternativa para el Nodo Professor

mantuvimos una relación cercana entre el esquema conceptual final y el original (el definido por el esquema ER), podemos obtener fácilmente una base de datos viable a partir de traducir el esquema ER a tablas relacionales.

Una vez que se dispone del conjunto de tablas, es necesario proveer un *retriever* para cada una de las entidades. El *retriever* que se utiliza con bases de datos relacionales utiliza el DSEL HaskellDB [LM99] para acceder a ellas mediante ODBC [ODB99]. Los aspectos generales de la instanciación de un *retriever* con HaskellDB se estudiaron en el Cap. 4.

Otras herramientas específicas son por ejemplo la función de construcción de especificación. La misma, a partir de un conjunto de entidades (que pueden recuperarse de una base de datos), construye una especificación de toda la aplicación (con sus contextos, nodos, UIs, etc.) lista para ser compilada a un *web site*. En el siguiente fragmento de código monádico (ver Apéndice A) ilustramos el uso de la función `academicSiteSpec` para generar una especificación; la función toma los conjuntos de entidades (`professors`, etc.), junto con las funciones de construcción de nodos (`professorNode`, etc.) y las correspondientes interfaces del usuario (`professorNodeUI`, etc.), retornando una especificación del *site*.

```
do
  ...
  spec <- academicSiteSpec
              professors      students      researchers ...
              professorNode   studentNode   researcherNode ...
              professorNodeUI studentNodeUI researcherNodeUI ...
```

El resultado de la invocación a `academicSiteSpec` es una especificación directamente compilable mediante una función apropiada.

Claramente no siempre es viable generar todo el *site* estáticamente. Ciertas herramientas, como el DSEL de manejo de CGI [vDM96] (ver Cap. 5), permiten generar algunas o todas las páginas en forma dinámica. Debido a que la generación dinámica es un aspecto puramente implementativo, no lo tratamos en detalle en este capítulo (en el Cap. 8 se muestra un ejemplo de la utilización del DSEL de CGI).

7.3.5 Refinamiento

Llegado a este punto, toda la biblioteca está completa en su versión básica. Claramente, la usabilidad de la biblioteca recién podrá comprobarse a partir de su uso real. Lo fundamental de la fase de refinamiento es la utilización de la biblioteca por varios grupos de desarrollo, atendiendo luego a las opiniones y requisitos de cada uno de los grupos. A partir de estas, podría refinarse la especificación de la biblioteca, y podría definirse además nuevas herramientas para facilitar el desarrollo. El refinamiento y extensión de las bibliotecas constituye un área importante para trabajo futuro.

7.4 Resumen

En este capítulo mostramos cómo la metodología DSEL puede aplicarse para facilitar el desarrollo de aplicaciones. Una biblioteca de autoría modela un dominio de aplicación particular, permitiendo reutilizar diseño y experiencia en las aplicaciones de ese dominio. Mostramos el ciclo general del desarrollo de bibliotecas de autoría, y mostramos cómo desarrollamos una biblioteca de autoría para *sites* académicos.

Capítulo 8

Un Ejemplo de Aplicación

En los capítulos anteriores nos hemos centrado en la presentación de las características principales de HyCom y en los enfoques posibles para el desarrollo de aplicaciones. Llegado este punto consideramos importante mostrar el desarrollo paso a paso de una aplicación real.

En este capítulo mostramos los pasos seguidos en el desarrollo de una aplicación académica mediante HyCom. Comenzamos presentando informalmente los requisitos de la aplicación; mostramos las decisiones que llevan a la utilización de una biblioteca de autoría y las implicaciones correspondientes; vemos cómo obtenemos un primer prototipo de la aplicación; finalmente mostramos los sucesivos refinamientos realizados sobre el prototipo hasta obtener la versión definitiva de la aplicación.

8.1 Presentación de la Aplicación

Las sistemas de información académicos son muy comunes en las universidades y facultades, cubriendo aspectos educativos y también relacionados con la investigación. Elegimos este dominio para ilustrar el desarrollo de una aplicación mediante HyCom principalmente debido a su amplia difusión. En esta sección detallamos informalmente los requisitos de la aplicación que desarrollaremos a lo largo del capítulo.

El *site* académico a desarrollar organiza la información del departamento de informática de una universidad. El departamento tiene profesores, estudiantes e investigadores. Además, ofrece diferentes carreras y aloja varios laboratorios. En los laboratorios se realizan las tareas de investigación. Los profesores dictan diferentes cursos, pertenecientes a una o más carreras. Cada carrera tiene un conjunto de cursos requeridos. Por cada alumno se tiene información sobre los cursos tomados y las carreras seguidas. El personal de investigación trabaja en uno o más laboratorios del departamento, y en una o más áreas de investigación; cada investigador está involucrado en proyectos pertenecientes a las áreas en las que trabaja. Por cada proyecto se tiene además una lista de las publicaciones realizadas

sobre el mismo.

Deseamos desarrollar un *site* que permita acceder a la información mencionada. Queremos además que la estructura de información del sistema esté bien definida y separada de los aspectos de presentación visual. Por otro lado, el sistema debe ser flexible y adaptarse en los cambios en los contenidos y estructura de la información. Finalmente, se desea poder obtener una primera versión funcional del sistema sin costos de desarrollo excesivamente elevados.

8.2 Aplicación de la Biblioteca de Autoría

Una vez establecidos los requisitos iniciales de la aplicación debemos comenzar a realizar los pasos de diseño ya vistos en capítulos anteriores. Sin embargo, vimos en el Cap. 7 que es posible reusar componentes y decisiones de diseño a través del uso de una biblioteca de autoría.

La utilización de una biblioteca de autoría comienza por el análisis del dominio de aplicación a desarrollar. Una biblioteca de autoría contiene un modelo conceptual del dominio de la aplicación realizado con un relativo grado de generalidad. El primer paso en el desarrollo consiste en analizar el grado de compatibilidad del modelo de la aplicación con el modelo de la biblioteca. Posteriormente, es necesario estudiar las adaptaciones que hay que realizar al modelo para ajustarlo a la utilización de la biblioteca.

En el ejemplo a desarrollar en este capítulo, elegimos un dominio de aplicación apto para la utilización de la biblioteca vista en el Cap. 7. Hicimos esto intencionalmente, a fin de ejemplificar simultáneamente el uso de una biblioteca en una aplicación y la definición de la misma.

8.3 Generando un Primer Prototipo

Adoptamos la biblioteca de autoría para *sites* académicos vista en el Cap. 7. En un principio, el modelo básico de la biblioteca es suficiente para generar un prototipo, por lo cual no necesitamos extenderlo.

La biblioteca de autoría incluye un esquema de bases de datos, preparado para el almacenamiento de los componentes del modelo conceptual. El primer paso que llevamos a cabo es introducir algunos datos de la aplicación en la base de datos. La tabla correspondiente a los profesores puede verse en la Fig. 8.1, con algunos datos de prueba. La Fig. 8.2 muestra la tabla de los cursos.

personID	personName	dateOfBirth	personAddress	personPhone	personPhoto
Smith00	Smith, Paul	10/10/70	49 Street, 123	750-5053	Smith00.Gif
Jones00	Jones, Simon	2/5/65	23 Street, 1390	728-4051	Jones00.Gif
Hughes00	Hughes, Peter	20/3/60	32 Street, 2137	849-5954	Hughes00.Gif
Graham01	Graham, James	18/2/69	80 Street, 2324	436-4463	Graham00.Gif

Figura 8.1: Tabla de Profesores con Datos de Prueba

courseID	courseName	courseDesc	courseCareer
0201	Functional Programming	A basic course on the funct	cs_lic
0302	Programming Language Concepts	An introduction to concepts	cs_lic
0301	Software Engineering	This course teaches the ba	cs_sa
0306	Object Oriented Software Engineering	In this course, we teach the	cs_lic

Figura 8.2: Tabla de Cursos con Datos de Prueba

Una vez que se tienen datos de prueba cargados es posible generar un prototipo en forma automática. HyCom provee funciones que permiten generar una especificación compilable automáticamente (ver Cap. 6). Suponiendo que utilizamos tal función y los *templates* de interfaz del usuario por defecto, podemos obtener un prototipo como se muestra en la Fig. 8.3 y en la Fig. 8.4. En general, al usar una biblioteca de autoría, el primer prototipo puede obtenerse sin necesidad de que el usuario escriba código HyCom (o escribiendo una cantidad mínima, necesaria para adaptar algún detalle menor).

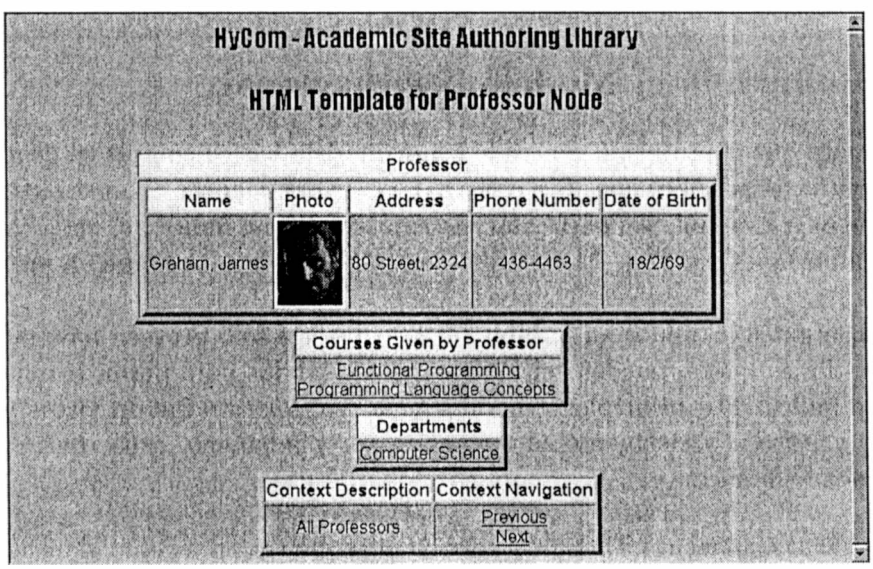

Figura 8.3: Primer Prototipo del Nodo de un Profesor

El prototipo generado cumple con la funcionalidad navegacional básica especificada por la biblioteca de autoría. Además, la interfaz del usuario es extremadamente simple. Sin embargo, el prototipo sirve al usuario para verificar si efectivamente la biblioteca de autoría sirve para su objetivo. Además, el usuario puede estudiar qué puntos en el modelo de la biblioteca deben extenderse para satisfacer los requisitos de la aplicación.

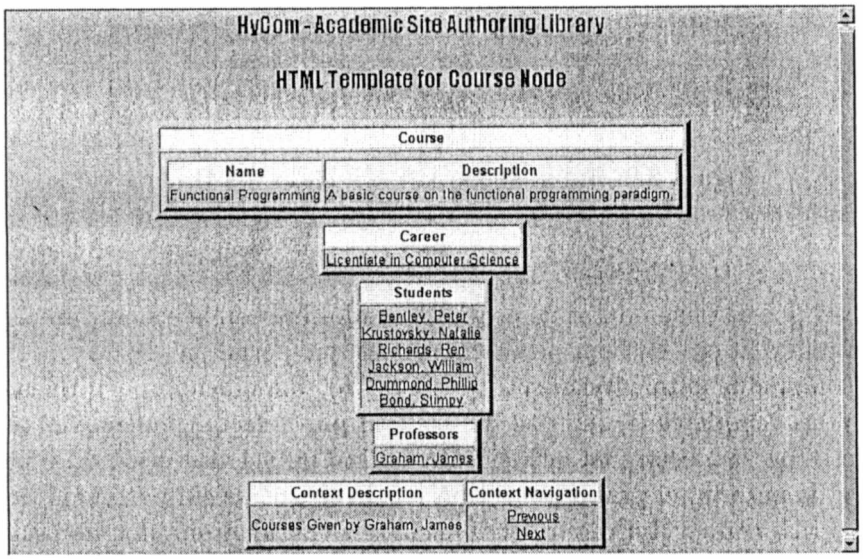

Figura 8.4: Primer Prototipo del Nodo de un Curso

8.4 Refinando el Modelo Navegacional

Supongamos que el prototipo generado con el modelo básico cumple en general con los requisitos de la aplicación. Sin embargo, deseamos realizar algunas extensiones al modelo navegacional. En particular, deseamos tener un índice por cada contexto, para facilitar la navegación. El esquema deseado se ilustra mediante un ejemplo en la Fig. 8.5.

Los contextos incluídos en la biblioteca de autoría sólo proveen navegación secuencial. Sin embargo, pueden ser fácilmente extendidos para permitir navegación mediante índice. Los principios generales para realizar esto fueron vistos ya en el Cap. 6 y consisten básicamente en instanciar apropiadamente cada contexto de la clase `IndexedContext`.

```
instance IndexedContext AllProfessors String where
   contextIndex (AllProfessors cs ns) = ContextIndex (zip ns cs)
```

Esta instanciación define la operación `contextIndex`, que permite obtener el índice del contexto en cuestión. El constructor `ContextIndex` tiene como parámetro una lista de pares, que en este caso tienen un `String` en el primer componente y un `ID` en el segundo. La función `zip`, estándar de Haskell, se utiliza para aparear la lista de `IDs` del contexto (`cs`) con la lista de los nombres de los profesores (`ns`). El resto de las instanciaciones se realiza de manera muy similar.

El siguiente paso es incluir el índice en la interfaz del usuario. Debido a que estamos realizando un prototipo, extenderemos la interfaz de forma simple, para incluir el índice al final de la página.

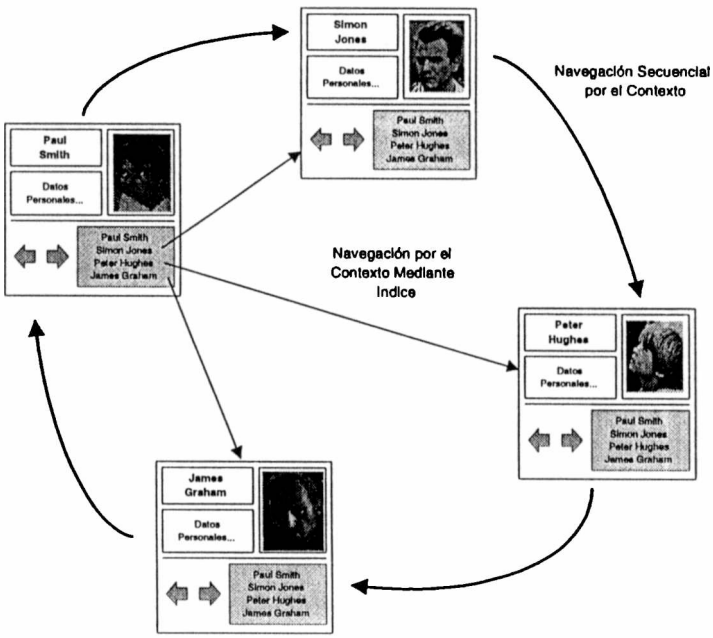

Figura 8.5: Esquema de Navegación Deseado

```
professorNodeUI2 professorUI
                 node
                 = setID (getID node) $
                   professorNodeUI professorUI node /=\
                   index
   where
      index = let ContextLinkIndex theIndex =
                     nodeContextLinkIndex nullID node
              in  setAlign Center $ setBorder 1 $ verticalBox $
                  map mkEntry theIndex

      mkEntry (s,l) = anchoredUI l (entryText s)
      entryText s   = setSize 4 $ setFont "Arial" $
                      textLabel s
```

Esta definición reutiliza la interfaz anterior (`professorNodeUI`), y la extiende agregándole un índice al final. La función `nodeContextLinkIndex` permite obtener un índice a partir del nodo (para más detalles ver Apéndice B). El resto de las funciones auxiliares ya fueron comentadas en ejemplos previos.

Una vez actualizada la definición de interfaz podemos obtener un nuevo prototipo de forma similar a como obtuvimos el primero. La página de un profesor incluyendo índices puede verse en la Fig. 8.6.

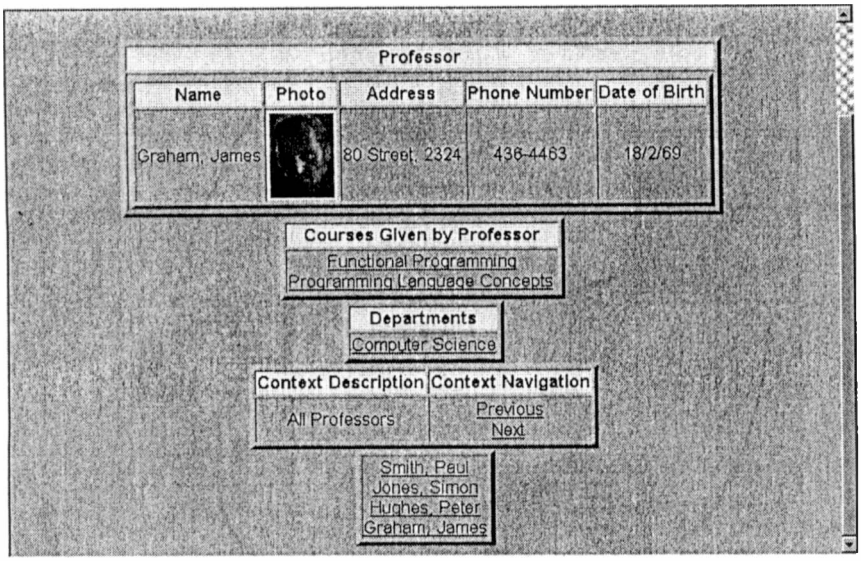

Figura 8.6: Nodo con Contexto Indexado

8.5 Refinando la Interfaz del Usuario

Una vez que la estructura navegacional satisface los requerimientos podemos focalizarnos en la definición de las interfaces del usuario de la aplicación. Si bien las definiciones básicas provistas por la biblioteca son suficientes para los prototipos iniciales, normalmente no lo serán para la aplicación definitiva. En general, la interfaz de una aplicación tiene ciertos requisitos estéticos (por ejemplo, un *look-and-feel* uniforme) o de usabilidad (la interfaz debe ser práctica para los usuarios potenciales).

La forma más simple de proveer una nueva interfaz es mediante el mecanismo de *templates*, explicado en el Cap. 5. Utilizando el mismo, el desarrollador puede usar una herramienta de edición de HTML para armar la estructura general de la página. La definición incorporará además un conjunto de tags específicos de HyCom, en donde se insertarán los componentes propios de cada nodo. Un ejemplo de *template* puede verse en la Fig. 8.7. Una página generada a partir del *template* mostrado puede verse en la Fig. 8.8.

Los *templates* proveen un mecanismo simple para la definición de interfaces del usuario. Sin embargo, puede obtenerse mayor control sobre la misma a partir de la utilización de combinadores y transformadores. Podemos utilizar esta opción para obtener una mejora sobre la versión anterior de la interfaz.

La definición de interfaz que mostramos a continuación es similar a otras vistas previamente. Utilizamos una función `standardUI` para capturar aspectos estándar de la visualización de los nodos (como por ejemplo, el *header* de las páginas, el fondo, etc.). Utilizamos también la función `mkAnchorsUI`, que ya vimos previamente y

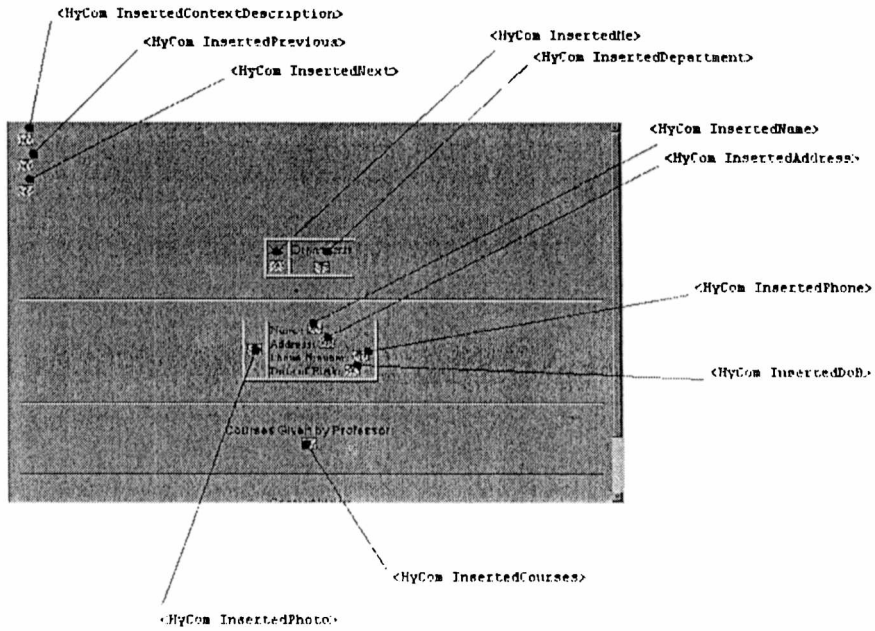

Figura 8.7: Un *Template* HTML

que sirve para mostrar una lista de *anchors*. Otras funciones utilizadas son `table` (permite armar una tabla), `tableRow` (permite armar una fila de una tabla), y el combinador `-##-` que permite secuenciar celdas en una fila de una tabla.

```
professorUI professorNode = standardUI $
                            personalInfo /=\
                            navigation   /=\
                            divisionLine /=\
                            courses      /=\
                            departments  /=\
                            index

   where

      personalInfo = setBorder 3 $
                     table $ tableRow $
                      name  -##- hSpace 50 -##-
                      photo -##- hSpace 50 -##-
                      data

      courses      =  coursesTitle /=\
                      mkAnchorsUI coursesAnchors

      coursesTitle =  setFont "Arial" $ setSize 5 $
                      textLabel "Courses"
```

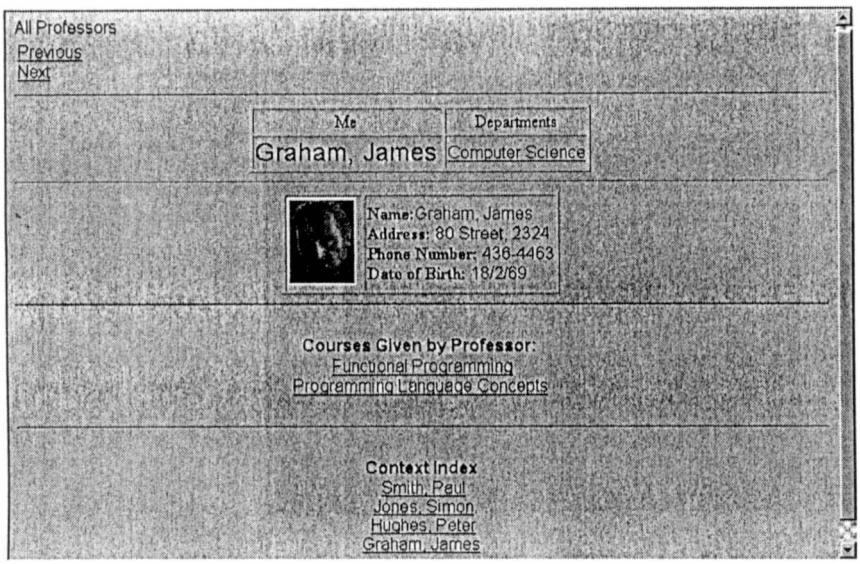

Figura 8.8: Interfaz Generada Mediante un *Template*

Una ventaja adicional de los combinadores y transformadores es su mayor claridad, obtenida gracias a su estilo declarativo. Además, las definiciones mediante este enfoque son independientes de la plataforma HTML. La interfaz definida puede verse en la Fig. 8.9.

8.6 Incluyendo Características de la Plataforma

Habiendo llegado a este punto del desarrollo disponemos de una aplicación cuyo esquema navegacional y su interfaz del usuario son satisfactorios. Sin embargo consideramos que la interfaz puede mejorarse incluyendo algunas características específicas de la plataforma de implementación.

8.6.1 Frames

A pesar de ser criticados, los *frames* permiten obtener ciertos efectos de interfaz que pueden ser importantes para nuestra aplicación. Para nuestro caso particular deseamos dividir la pantalla en dos *frames*, ubicando el contenido en el *frame* derecho y el índice del contexto en el *frame* izquierdo.

HyCom provee componentes para manejar *frames* con practicidad, como se vió en el Cap. 5. Un *frame* es un componente de interfaz que permite visualizar el contenido de dos nodos en forma simultánea. Un *frame* visualizará al nodo de contenido, y otro *frame* a un nodo conteniendo el índice del contexto.

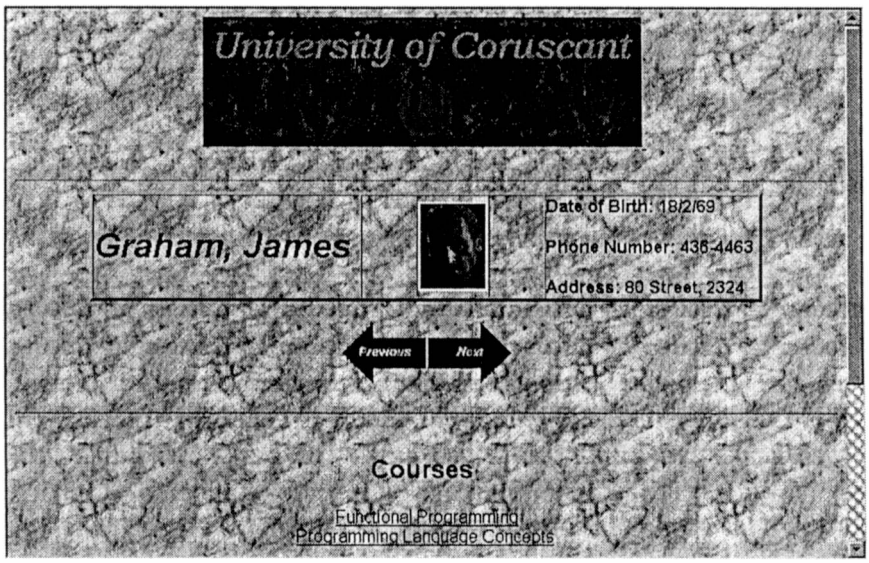

Figura 8.9: Interfaz Generada Mediante Combinadores

La primera tarea a resolver consiste en la definición de un nodo índice. Consideramos que un nodo de este tipo no resulta en una alteración al modelo navegacional, sino en una consideración de interfaz. Esto es porque el agregado de este nodo no define una nueva forma de navegación, sino una forma diferente de visualizar el índice. Por lo tanto, definimos cada nodo de este tipo sólo a nivel de la interfaz.

```
allProfessorsIndexNode = theTitle     /=\
                         divisionLine /=\
                         theDesc      /=\
                         theIndex     /=\
                         divisionLine

   where theTitle = setSize 5 $ setFont "Arial" $
                    textLabel "Index"
         theDesc  = setSize 3 $ setFont "Arial  $
                    getContextDescription $
                    allProfessorsContext
         theIndex = contextIndexUI allProfessorsContext
```

La función `contextIndexUI` define la visualización estándar para este tipo de nodos. La misma puede definirse mediante *templates* o combinadores, como en los otros casos de interfaces que ya vimos. Es importante ver que los *links* en el índice que afecten al otro nodo deben transformarse mediante la función `hasTargetID` (la misma es análoga a `HasTarget`, pero utiliza sólo el `ID` del *frame* destino). La función `hasTargetID` transforma un *link*, de forma que la navegación se efectúe en otro *frame* y no en el que aloja al *anchor*.

```
contextIndexUI c = setID (getID c) $
                       let ContextLinkIndex theIndex =
                             contextLinkIndex c nullID
                       in  verticalBox $ map mkEntry theIndex

   where mkEntry (s,l) = hasTargetID theFrameID $
                         anchoredUI l (textLabel s)
         theFrame      = iD "Content Frame"
```

El siguiente paso es construir el nodo que aloja al *frame*. Este nodo actúa como una ventana a través de la cual pueden verse simultáneamente el nodo índice y el nodo con el correspondiente contenido. La función `frameSet` permite construir una interfaz en la cual coexisten varios *frames*, a modo de ventanas superpuestas (ver el Cap. 5). La función toma una lista de los *frames* y un parámetro adicional (`VFrame` o `HFrame`) que indica si los mismos se agrupan horizontalmente o verticalmente. Cada *frame* a su vez se construye aplicando el transformador `frame` a cualquier componente de interfaz.

```
allProfessorsFrame node = frameSet [ indexFrame,
                                     contentFrame ] VFrame
   where indexFrame   = setID "Index Frame" $
                        frame allProfessorsNodeIndex
         contentFrame = setID "Content Frame" $
                        setFrameSize 5 $ frame $
                        professorNodeUI node
```

El proceso se realiza en forma similar para el resto de las interfaces. La interfaz resultante puede verse en la Fig. 8.10.

En general, la utilización de *frames* implica tener en cuenta algunos detalles de implementación. Por ejemplo, es viable querer descartar los *frames* bajo ciertas condiciones de la navegación (por ejemplo, al seguir un *link* que va fuera de la aplicación). De la misma manera, es posible que diferentes partes de la aplicación manejen los *frames* de distinta forma. Estos detalles pueden resolverse fácilmente, dependiendo de los conocimientos técnicos del desarrollador; no los analizamos aquí para preservar la simplicidad del ejemplo.

8.6.2 Aspectos Dinámicos

La mayoría de los *sites* modernos utilizan en mayor o menor grado ciertos aspectos de generación dinámica de páginas. En general, los mismos se usan para realizar consultas sobre el *site*, efectuar búsquedas en un índice, etc. Para nuestro ejemplo deseamos agregar la posibilidad de buscar profesores que dicten determinado curso.

Comenzamos construyendo el *form* que le permite al usuario expresar su consulta. En nuestro caso particular deseamos que el mismo contenga un campo de texto en donde ingresar el nombre del curso y botones para reiniciar el campo y efectuar la consulta. Además, el *form* debe estar parametrizado con el *script* CGI que efectúa efectivamente la consulta (asumimos que el mismo es `search.cgi`).

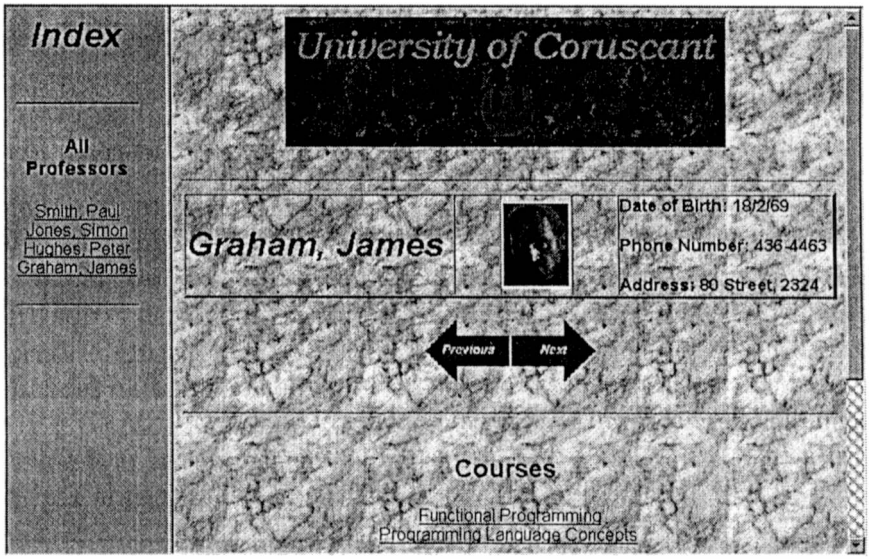

Figura 8.10: Utilización de *Frames*

```
searchForm = form action $
               textLabel "Enter Course Name:"        /=\
               (setID (iD "Name") $ textField "" 64) /=\
               resetButton <=< submitButton

   where action = FormAction
                    Get
                    "http://ourserver/cgi-bin/search.cgi"
```

El siguiente paso consiste en la definición del *script* de búsqueda en sí. HyCom permite construir *scripts* CGI en forma sencilla, mediante la integración con el DSEL *CgiLib* [vDM96].

El siguiente código recibe un *query* del ambiente (el DSEL CgiLib nos permite ignorar si el *query* se recibió por el *command line* o por variables de *environment*), y obtiene el valor para el campo "Name" (observemos que el nombre del campo coincide con el `ID` del campo de texto del *form*). Utilizando el mismo, recupera el código del curso mediante el *retriever* apropiado. Utilizando el código del curso recupera los profesores que dictan ese curso. Finalmente, retorna los resultados de la búsqueda, formateados mediante una función `resultUI` cuya definición no mostramos.

```
searchScript query
   = do
        name       <- return $ find query "Name"
        courses    <- retrieveBy courseRetriever (condC name)
        professors <- retrieveBy professorRetriever (condP courses)
```

```
        return $ resultUI professors

where condC cName c = (getCourseName c) == cName
      condP cs p    = (empty cs) `or`
                      ((getID $ head cs) `in` (getCoursesGiven p))
```

Una vez realizados los dos puntos anteriores, la mayor parte del problema ha sido resuelta, quedando sólo aspectos menores por resolver. Por un lado, deberemos incluir el *form* en la interfaz de los nodos de los profesores. Por otro lado, deberemos crear el archivo `search.cgi` que ejecute el *script* definido más arriba (la forma de hacer esto depende del sistema operativo en el cual se ejecute el *server*). No comentamos estos aspectos en más detalle debido a que no resultan fundamentales para el ejemplo.

El *form* anterior, insertado en la interfaz del usuario de los profesores, puede verse en la Fig. 8.11. El resultado de una consulta puede verse en la Fig. 8.12.

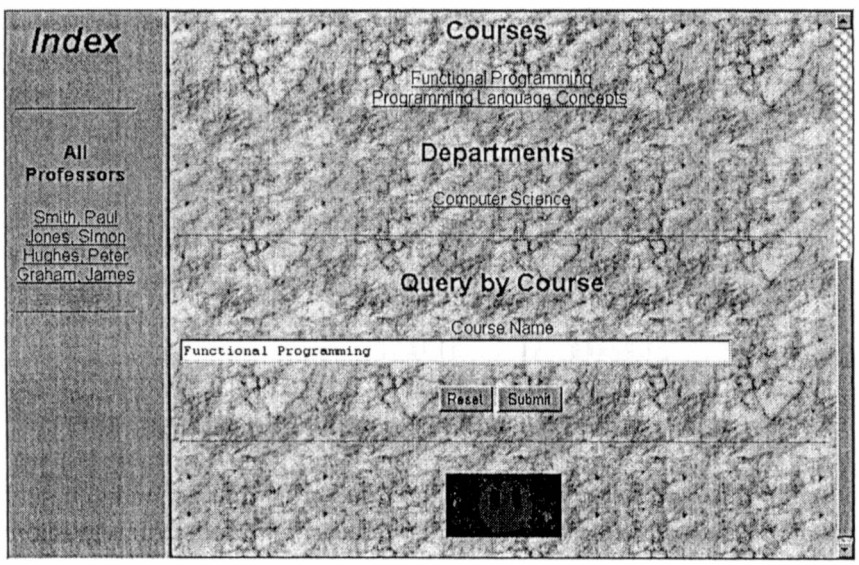

Figura 8.11: Interfaz con un *Form*

8.7 Resumen

En este capítulo mostramos, en forma resumida, los pasos involucrados en la construcción de una aplicación real mediante HyCom y una biblioteca de autoría. Comenzamos esbozando los requerimientos de la aplicación, consistente en un *site* académico. Luego analizamos las condiciones de la aplicación de la biblioteca. A continuación refinamos aspectos referentes a la navegación y a la interfaz del usuario, para suplir las necesidades de la aplicación. Finalmente vimos cómo incorporar aspectos específicos de la plataforma WWW, como *frames* y *scripts* CGI.

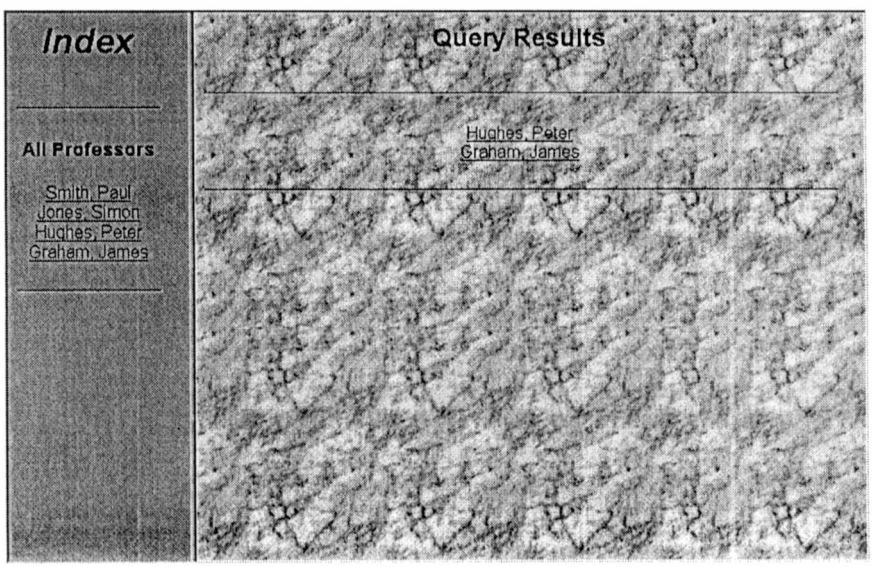

Figura 8.12: Resultado de una Consulta

Capítulo 9

Uso de HyCom con un Método Existente

Si bien existen varias metodologías para el diseño de hypermedia, generalmente el desarrollo final de la aplicación implica perder de vista los elementos de alto nivel de los modelos. En general la implementación no refleja las primitivas presentes en las metodologías.

En este capítulo mostramos cómo HyCom puede utilizarse en conjunción con una metodología muy difundida, ganándose claridad en el mapeo de las primitivas de la metodología a la implementación.

9.1 Conceptos Básicos de RMM

La metodología RMM (*Relationship Management Methodology*) es frecuentemente citada como la primer metodología para el diseño de hypermedia. Presentada por primera vez por Isakowitz et al. en 1995 [ISB95], ha pasado por varias ampliaciones desde entonces. La versión que utilizamos aquí es la presentada en [IKK98]. A lo largo del resto de esta sección realizamos una revisión de los conceptos básicos de RMM, y presentamos el ejemplo en el cual trabajaremos a lo largo del capítulo. Asumimos un conocimiento básico de RMM, por lo cual los lectores sin conocimiento previo de la metodología pueden consultar los trabajos [ISB95, IKK97a, IKK97b, IKK98].

El método RMM está basado en el conocido modelo de entidades y relaciones (ER) [EN90]. El usuario comienza el desarrollo definiendo un modelo conceptual de la aplicación, utilizando entidades y relaciones. Esto constituye el primer paso del método. Un esquema ER simple se muestra en la Fig. 9.1.

Los siguientes dos pasos involucran realizar un diseño *top-down* y *bottom-up* de la estructura navegacional de aplicación. El objetivo es la obtención del *diagrama de aplicación*. Un diagrama de aplicación consiste en M-Slices y *links* entre ellos. Las *M-Slices* son unidades presentacionales, y consisten en atributos de una o más entidades (en particular, cada M-Slice tiene una entidad *owner* o "dueña"), *anchors*,

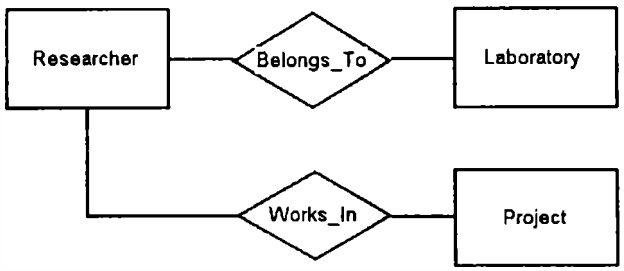

Figura 9.1: Un Diagrama ER

estructuras de acceso (como índices o *guided tours*), y otros M-Slices. Un M-Slice es una representación abstracta de una porción de información que va a ser mostrada en la aplicación. Los M-Slices de nivel superior (*top-level*) constituyen los nodos de la aplicación.

En la Fig. 9.2 mostramos un diagrama de aplicación correspondiente al diagrama ER de la Fig. 9.1. Aquí tenemos tres M-Slices *top-level* y *links* entre ellos. La definición completa de los M-Slices `top` y `name` de la entidad Researcher se muestra en la Fig. 9.3. El M-Slice *top* tiene algunos atributos de su dueña (la entidad Researcher): *rank* y *photo*. También incluye otro M-Slice de su dueña, el M-Slice *name*. La parte inferior del M-Slice incluye elementos que no corresponden a la entidad dueña, como el logo de la entidad Laboratory, que sirve como *anchor* para el M-Slice *top* de la entidad Laboratory. También incluye un índice de proyectos, anclado en el M-Slice *name* de la entidad Project (de hecho, este es un tipo particular de M-Slice con un único atributo).

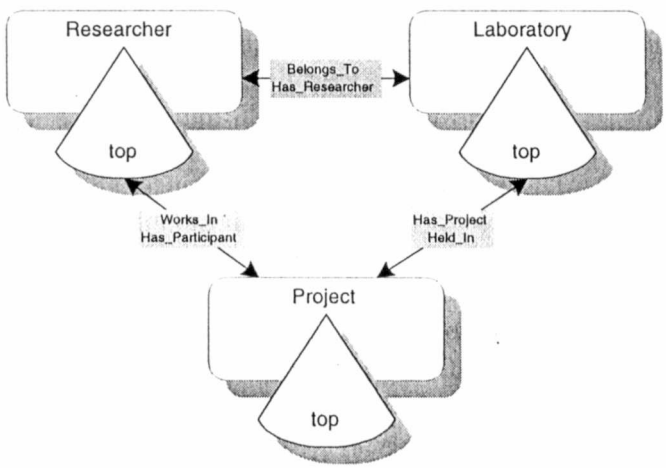

Figura 9.2: Un Diagrama de Aplicación

El método define cuatro fases más. Sin embargo, estas fases no han sido de-

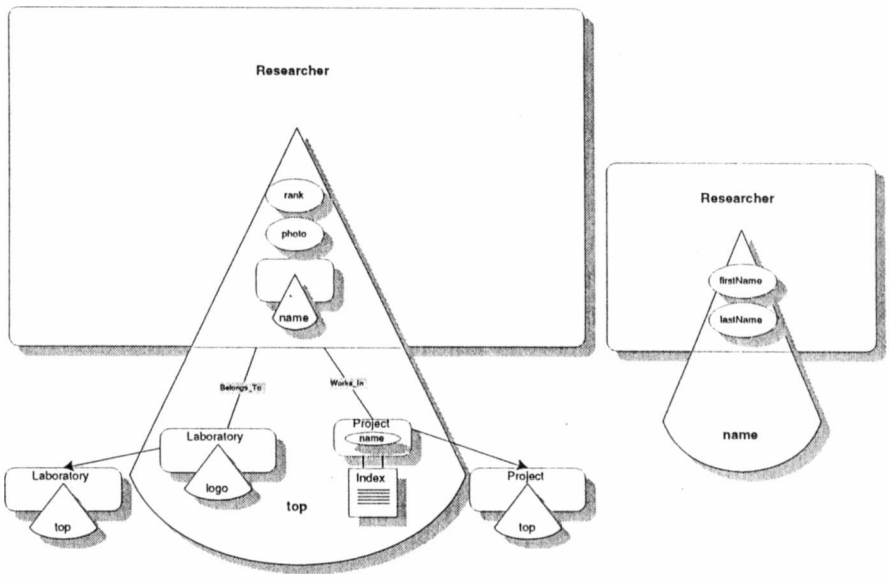

Figura 9.3: M-Slices `top` y `name` Pertenecientes a la Entidad Researcher

finidas formalmente por sus autores, y generalmente, los desarrolladores toman enfoques *ad-hoc* para estas fases [BBI98]. Consecuentemente, nos focalizaremos en las fases mencionadas, que constituyen la parte fundamental del método.

9.2 Mapeo de RMM a HyCom

Utilizando el ejemplo presentado previamente mostramos cómo desarrollar un mapeo posible de las características de RMM a HyCom.

El primer paso es simple e involucra representar el modelo ER con un conjunto de tipos. Existen muchas formas de hacer esto, y aquí mostramos la más simple y directa.

```
data Researcher = Researcher { firstName :: String,
                               lastName  :: String,
                               rank      :: String,
                               photo     :: String,
                               intro     :: String,
                               projects  :: [Project],
                               lab       :: Laboratory }
```

En este ejemplo decidimos incluir las relaciones en las entidades, debido a que las relaciones no poseen atributos. Esto no es necesariamente el caso, y las relaciones también pueden representarse mediante tipos de la misma manera que las entidades. También decidimos hacer componentes a las entidades (esto puede hacerse fácilmente proveyendo definiciones para las operaciones de la clase `Component`).

El desarrollador normalmente debe manejar conjuntos de entidades, generalmente almacenadas en una base de datos. Esto puede modelizarse instanciando la clase `Retriever` para cada entidad con un *retriever* apropiado.

El punto más interesante del mapeo consiste en la representación de *slices* (a partir de ahora, utilizaremos los términos M-Slice y *slice* de forma equivalente). Queremos expresar los *slices* claramente, pero también aprovechar el chequeo de tipos para evitar la definición de *slices* incorrectos. Lo primero que hacemos es definir una clase de tipos `Owned`, para restringir los componentes de los *slices*. La clase `Owned` tiene información de tipo sobre la entidad dueña, y especifica la responsabilidad de poder recuperar la entidad dueña.

```
class (Component o) => Owned a o where
   owner :: a -> o
```

Ahora podemos ocuparnos de la representación de *slices*. Los *slices* tienen una entidad dueña, un conjunto de componentes pertenecientes a la entidad dueña, y un conjunto (posiblemente vacío) de componentes no pertenecientes a la dueña. Definimos entonces el tipo `Slice` para mantener la información necesario de un *slice*.

```
data (Component owner, Owned a owner) =>
   Slice a b owner = Slice a b owner ID
```

El constructor de tipos `Slice` tiene tres parámetros. Nótese que el contexto (`Owned a owner`) restringe la combinación de tipo a a tener el mismo tipo de dueña que la dueña del *slice* que se está definiendo (dada por el parámetro `owner`). Nótese también que el constructor `Slice` (del lado derecho de la declaración) también tiene lugar para un `ID`, que es necesario para hacer a `Slice` una instancia de la clase `Component`. Realizando una comparación con la notación gráfica de *slices*, podemos ver que la combinación restringida (de tipo `a`) corresponde a la parte superior del *slice*, y que la otra combinación (de tipo `b`) corresponde a la parte inferior del *slice*.

Algunos *slices* sólo incluyen datos de su entidad dueña. Para este tipo particular de *slice*, definimos el tipo `PureSlice`. El mismo es similar al tipo `Slice`, excepto por el hecho de que no mantiene la combinación de componentes que no pertenecen a la entidad dueña.

```
data (Component owner, Owned a owner) =>
   PureSlice a owner = PureSlice a owner ID
```

Otras representaciones interesantes de conceptos de RMM son los tipos `Att` (modela M-Slices con un único atributo), `RMM_Anchor` (que modela *anchors* dentro de los *slices*) y `RMM_Index` (que modela índices). No mostramos las definiciones de estos componentes aquí. Todos estos tipos son instancias de la clase `Owned`, permitiendo evitar la formación de estructuras inválidas.

Hemos mencionado previamente que un *slice* mantiene combinaciones de componentes, perteneciendo o no a la entidad dueña. Las combinaciones son modeladas

con *combinadores* (el concepto principal subyacente en HyCom). Proveemos dos tipos de combinaciones: combinaciones de componentes de la misma entidad dueña (`ownerCombined`) y combinación de componentes en general (`sliceCombined`). Estos combinadores son modelados con tipos y funciones para crear objetos de estos tipos.

```
data (Component o, Owned a o, Owned b o) =>
   OwnerCombined a b o = OwnerCombined a b

ownerCombined,(+--+) :: (Owned a o, Owned b o) =>
                        a -> b -> OwnerCombined a b o

data SliceCombined a b = SliceCombined a b

sliceCombined,(-&&-) :: a -> b -> SliceCombined a b
```

Estos combinadores previenen el hecho de que el usuario combine componentes pertenecientes a la dueña del *slice* con componentes que no pertenecen a la dueña. Nótese que en `ownerCombined` utilizamos contextos en una forma similar que en la declaración de `Slice`: los contextos de clase `(Owned a o)` y `(Owned b o)` aseguran que componentes de diferentes dueñas no sean combinados, y que la combinación resultante también pertenezca a la entidad dueña correcta. Nótese también que hemos provisto definiciones infijas de los combinadores mencionados.

Algunas funciones interesantes son aquellas utilizadas para construir *anchors*, índices y *slices* con un único atributo.

```
rmmAnchor :: (HLComponent l) => a -> l -> RMM_Anchor a l

rmmIndex  :: (Component owner, HLComponent l, Owned a owner) =>
               (owner -> a) -> (owner -> l) -> [owner] ->
                o -> RMM_Index a l o

att       :: (c -> a) -> c -> Att a c
```

Estamos en condiciones de mostrar la definición en HyCom del M-Slice *top*, perteneciente a Researcher. Definimos la función `researcherTop`, que toma una entidad `researcher`, una entidad `lab` y una lista `projects`, retornando el M-Slice *top*. Observemos que utilizamos las funciones `att` (para construir atributos que contienen un valor e información sobre su entidad dueña) y `linkTo` (utilizada para construir un *link* dado un *slice* destino).

```
researcherTop researcher lab projects
   = slice fromOwner other researcher

   where fromOwner = (att rank researcher)       +--+
                     (att photo researcher)      +--+
                     (researcherName researcher)
```

```
other      = (rmmAnchor (laboratoryLogo lab) linkToLab)
             -&&-
             (rmmIndex projectNameAtt
                       linkToProject
                       projects
                       researcher)

linkToLab       = linkTo (laboratoryTop lab)
linkToProject p = linkTo (projectTop p)
```

Comparando el código resultante con la notación gráfica del *slice* en cuestión, notamos que la versión HyCom no ha perdido la expresividad original. La declaración local `fromOwner` corresponde a la parte superior de la notación gráfica del *slice*, y la declaración `other` pertenece a la parte inferior. Además, utilizando tipos evitamos que el usuario pueda definir *slices* mal formados.

El siguiente paso consiste en definir cómo el M-Slice va a visualizarse en la interfaz del usuario. Esto puede hacerse mediante componentes y combinadores de UI. Definimos una función que toma un M-Slice `researcherTop` y retorna un componente de UI. Asumimos que tenemos funciones para recuperar los componentes internos del *slice*, como por ejemplo el M-Slice `name`, el atributo `rank`, etc. (estas funciones pueden definirse fácilmente por *pattern matching*). También hacemos uso de otros componentes de UI, como `rmmIndexUI` (el componente de UI para índices), `researcherNameUI` (el componente de UI para el *slice* `name`) y `divisionLine`, para los cuales no mostramos las definiciones.

```
researcherTopUI researcherTop =  header              /=\
                                 personalData        /=\
                                 researchInterests
  where
    header            = laboratoryLogoUI logo /=\
                        divisionLine
    personalData      = (profile <=< image photo)  /=\
                        divisionLine
    profile           = researcherNameUI nameSlice /=\
                        textLabel rank
    researchInterests = (textSize 4 (textLabel "Projects")) /=\
                        (rmmIndexUI projectIndex)
    nameSlice         = getNameSlice researcherTop
    projectIndex      = getProjectIndex researcherTop
    logo              = getLogoSlice researcherTop
    photo             = getAttrContent (getPhotoAttr researcherTop)
    rank              = getAttrContent (getRankAttr researcherTop)
```

Una vez más, nótese que incluso al definir la UI las construcciones de diseño originales (*slices*, atributos) están aún presentes. El componente de UI resultante puede compilarse, resultando en un prototipo en HTML u otra plataforma, como

se explicó en otros capítulos. La página HTML prototípica puede verse en la Fig. 9.4. Hemos marcado las representaciones de UI de las construcciones RMM, para facilitar la comparación con el código HyCom.

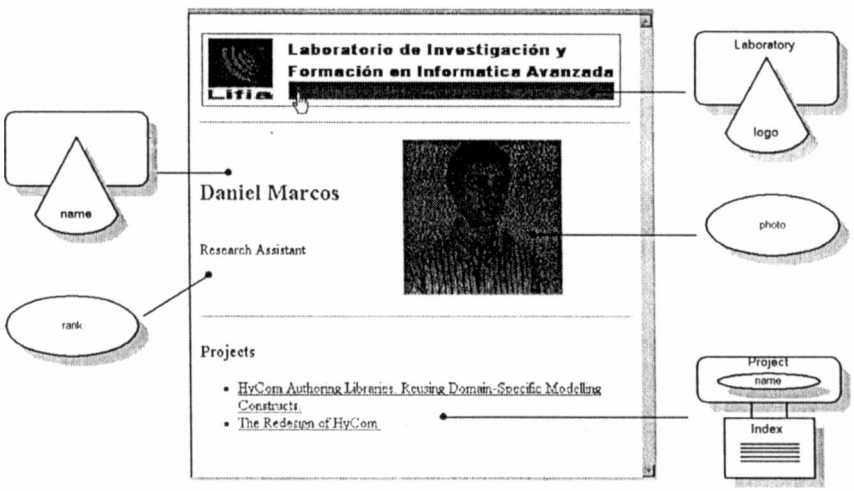

Figura 9.4: Página WWW Correspondiente al M-Slice `top`

9.3 Resumen

Existen diversas metodologías muy difundidas para el diseño de hypermedia. Frecuentemente, los principios de alto nivel de las metodologías se pierden al realizar la implementación, perdiéndose sus ventajas originales.

En este capítulo mostramos cómo HyCom puede utilizarse en conjunción con la metodología RMM. El mapeo resultante no sólo preserva la claridad y expresividad del modelo original, sino que aprovecha el chequeo de tipos para verificar la validez de las construcciones definidas.

Parte IV

Comentarios Finales

Capítulo 10

Trabajos Relacionados

Existen varios trabajos con enfoques y objetivos similares a los de HyCom. Los diferentes enfoques tienen distintos puntos de contacto en cada caso.

En este capítulo hacemos una breve reseña de los trabajos relacionados que consideramos más relevantes por su relación con HyCom.

10.1 WebComposition y WCML

10.1.1 Conceptos Básicos

El enfoque de WebComposition [GWG97, GBG98] se basa en proveer un modelo orientado a objetos para la construcción de aplicaciones WWW. De acuerdo a este modelo, una aplicación se descompone jerárquicamente en componentes. En los niveles más altos de la jerarquía un componente puede modelar una página o incluso un *site*. Bajando en el nivel de la jerarquía un componente modela partes de una página, como tablas, *anchors*, etc. Los componentes de este nivel pueden modelar también abstracciones específicas de la aplicación para la organización de las páginas, como por ejemplo una columna de anunciantes, un índice, etc. Las hojas en la jerarquía de componentes se denominan primitivas, mientras que los otros componentes son llamados *composites*. El modelo no prescribe la granularidad de las primitivas, quedando este aspecto librado al diseñador.

Un componente está definido por su estado y su comportamiento. El estado de un componente está definido por una lista de propiedades. Las propiedades son simplemente pares nombre-valor. El valor puede ser un escalar o un *array*, y puede contener tanto el valor mismo como una referencia a otro objeto. Las propiedades modelizan, en general, atributos de HTML. Por ejemplo, un componente tabla tendrá propiedades como borde, espaciado, valores por defecto, etc.

El comportamiento de un componente está definido por operaciones sobre su estado. Las operaciones soportadas por los componentes de WebComposition se reducen a la inspección y modificación de propiedades, y a la generación de código HTML.

Los conceptos do WebComposition se implementan en un lenguaje de marcas, basado en XML, denominado WebComposition Markup Language (WCML). El modelo de objetos de WCML está basado en prototipos y no en clases (de forma similar a Self [U+87]), siendo este modelo más adecuado para trabajar con componentes de la forma que lo hace WebComposition.

Finalmente, los diseñadores de WebComposition planean construir un *repositorio de componentes*. El repositorio (aún no implementado en su totalidad) consistiría en un conjunto de componentes reusables, cubriendo distintos aspectos del desarrollo de aplicaciones.

10.1.2 Comparación

El enfoque de WebComposition/WCML persigue, en algunos aspectos, objetivos similares a los de HyCom. El principal punto de contacto está dado por la intención de reducir la distancia entre la implementación y los modelos de ingeniería. WebComposition propone la utilización de WCML para realizar el mapeo, mientras que nuestro enfoque propone realizarlo con HyCom.

Nuestra opinión es que el modelo de objetos de WCML es pobre. Los componentes soportan funcionalidad muy limitada, con lo cual aspectos complejos en la generación de una aplicación (por ejemplo, calcular los nodos a partir de una base de datos) no pueden realizarse directamente en WCML y debe recurrirse a un lenguaje auxiliar. Por el contrario, el enfoque DSEL de HyCom permite que todos los aspectos de la aplicación se manejen en un entorno homogéneo, dado por el lenguaje huésped. HyCom tiene acceso directo a las características de Haskell, que es computacionalmente completo, y además se integra inmediatamente con todos los DSELs existentes embebidos en Haskell.

Otro punto de contacto con HyCom es el repositorio de componentes. De cierta forma, este repositorio se relaciona con el concepto de biblioteca de autoría, que consiste en reusar componentes y elementos de diseño específicos al dominio. Las bibliotecas de autoría se diferencian del enfoque de WebComposition debido a que no sólo proveen componentes reusables, sino que también proveen una arquitectura abstracta de la aplicación a desarrollar.

Fuera de las diferencias obvias entre HyCom y WebComposition/WCML (como el paradigma adoptado y la forma de implementación) existen otras. El lenguaje WCML no provee chequeo de tipos, lo cual permite la posibilidad de que se escriban componentes incorrectos respecto al modelo o metodología utilizado en el diseño. HyCom en cambio puede beneficiarse del chequeo de tipos para garantizar que gran parte de los errores del modelo se detecten en tiempo de compilación. Por otro lado, WCML está basado exclusivamente en la WWW, mientras que HyCom es independiente de plataforma.

10.2 OOHDM-Web

10.2.1 Conceptos Básicos

El ambiente de desarrollo OOHDM-Web [SdAP99] fue diseñado para el desarrollo de aplicaciones dinámicas en la WWW, tomando la metodología OOHDM como base. La idea central de este enfoque consiste en la traducción de un modelo OOHDM a un conjunto de tablas relacionales, *templates* de páginas HTML y *scripts* en el lenguaje Lua [HBI97].

La metodología OOHDM involucra varios pasos, los cuales se vieron en forma resumida en el Cap. 2. OOHDM-Web se focaliza en el diseño navegacional, el diseño de interfaz y la implementación en sí.

El desarrollo comienza con un diseño navegacional definido en OOHDM, consistente en un modelo de aplicación orientado a objetos enriquecido con primitivas de navegación como contextos e índices. OOHDM-Web plantea la traducción de este modelo a un conjunto de tablas, siguiendo una heurística extremadamente simple. Los contextos navegacionales e índices también son traducidos a tablas de forma trivial.

La siguiente fase consiste en el diseño de interfaz. OOHDM-Web plantea la realización de *templates* HTML, que definen la estructura básica de visualización de cada uno de los nodos.

La implementación de la aplicación consiste en integrar las tablas relacionales y los *templates* utilizando el lenguaje de *scripting* Lua. Básicamente, se utilizan *scripts* Lua que realizan el acceso a la base de datos, y estos se embeben en los *templates* HTML. El resultado es un conjunto de páginas con código Lua insertado, que generan páginas HTML automáticamente bajo demanda. Este tipo de arquitectura es básicamente muy similar a la que se logra con *servers* de aplicaciones WWW como ColdFusion [Col99] o tecnologías relacionadas como ASP [ASP99].

10.2.2 Comparación

El punto de contacto entre OOHDM-Web y HyCom está dado por el objetivo común de integrar la implementación con los modelos de alto nivel. HyCom permite describir una aplicación con un alto nivel de abstracción, o incluso traducir un modelo realizado con una metodología determinada, sin perder expresividad. OOHDM-Web persigue un objetivo similar a través de la traducción de un modelo OOHDM a un conjunto de tablas relacionales, *templates* HTML y *scripts* Lua.

Nuestra opinión es que el enfoque de OOHDM-Web no resulta en una notable ganancia a nivel conceptual. La traducción de las primitivas OOHDM directamente a tablas relacionales resulta en una notable pérdida de riqueza con respecto al modelo original. En este sentido, HyCom permite expresar diseños con una riqueza mucho mayor (como en el caso mostrado en el Cap. 9).

El ambiente OOHDM-Web representa una alternativa interesante si el énfasis del desarrollo está en aspectos implementativos. Por ejemplo, si el interés principal

es tener acceso a una base de datos y generación dinámica de páginas, OOHDM-Web presenta un enfoque de más alto nivel que muchos productos comerciales. Por el contrario, HyCom plantea tener un nivel más de abstracción entre los aspectos de implementación (como el acceso a una base de datos) y los modelos de ingeniería. Ese nivel extra está dado precisamente por la descripción de los modelos en HyCom, y actúa como conciliador entre la implementación y el modelo de ingeniería.

10.3 PageJockey

10.3.1 Conceptos Básicos

El ambiente PageJockey fue presentado originalmente en [FNN96] como un ambiente orientado a objetos para el desarrollo de hypermedia en forma incremental. Los puntos fundamentales en los que se basa PageJockey son el soporte al desarrollo mediante prototipos, y la creación de *templates* reusables.

PageJockey persigue una serie de objetivos en torno a su puntos fundamentales. Sus autores plantean que un entorno de autoría ideal debe permitir separar el contenido de la aplicación de la forma de presentación, a la vez que permita abstraer y reusar estructuras. Además, ese ambiente debe permitir la generación automática de prototipos, para permitir la evaluación temprana de los diseños. El entorno PageJockey es precisamente una implementación de un ambiente tomando esas necesidades como base.

Un desarrollo con PageJockey comienza por la identificación de los datos del dominio (se asumen que los mismos se encuentran bien estructurados, como producto de una fase de modelado conceptual realizada previamente). El siguiente paso consiste en la definición de la estructura de la hypermedia a partir de la elaboración de *templates*, que abstraen diferentes aspectos de la hypermedia (por ejemplo, estructura de navegación, *templates* de interfaz, etc.). El desarrollo continúa a partir del refinamiento sucesivo de los *templates*, con un intenso *feedback* producido por la prototipación, hasta llegar a la aplicación final.

10.3.2 Comparación

Los puntos de contacto entre HyCom y PageJockey se relacionan principalmente con la prototipación y el reuso de estructuras.

En primer lugar, tanto HyCom como PageJockey plantean un ciclo de producción similar. Ambos proveen facilidades de prototipación automática, que permiten adoptar un enfoque iterativo en el desarrollo. Los prototipos generados por PageJockey sin embargo están en un formato específico del ambiente, mientras que HyCom puede generar prototipos para cualquier plataforma para la que se provea un módulo de compilación.

La reutilización de componentes es otro punto de similitud entre nuestro enfoque y PageJockey. HyCom permite capturar estructuras reusables a través de funciones, que cumplen el rol de componentes genéricos instanciables (debemos destacar que

las funciones se encuentran en un nivel de abstracción mayor que los componentes). PageJockey también fomenta la creación de *templates*, pero se basa en un modelo orientado a objetos basado en prototipos, similar al de Self [U+87].

Es importante destacar una diferencia fundamental entre los dos enfoques. HyCom se presenta como un lenguaje de autoría, mientras que PageJockey tiene la forma de un ambiente de desarrollo interactivo. En este sentido, HyCom no sólo persigue los objetivos relacionados con el ciclo de desarrollo que mencionamos (prototipación, reuso), sino que también propone una notación formal para expresar los diseños.

Capítulo 11

Trabajo Futuro

Claramente, la utilización de un DSEL para el desarrollo de hypermedia es un enfoque novedoso y joven. El éxito de HyCom en aplicaciones reales depende de seguir trabajando en él. Distinguimos las siguientes como las líneas de trabajo futuro más relevantes.

11.1 Soporte para Nuevas Plataformas

HyCom es, en principio, independiente de plataforma, y puede traducirse a cualquier plataforma de implementación para la cual se provean funciones de traducción. De la misma forma que mostraba la plataforma HTML en esta tesis, pensamos desarrollar soporte para otras, como XML y aplicaciones en Visual Basic.

El soporte para otras plataformas presenta varios desafíos interesantes. Por un lado, implica traducir efectivamente las construcciones de HyCom a la plataforma en cuestión. Por otro lado, exige el soporte de características específicas de la plataforma dentro del marco de HyCom, sin perder su esencia.

11.2 Bibliotecas de Autoría

En esta tesis se presentó una biblioteca de autoría para *sites* académicos. Actualmente estamos ocupados en el desarrollo de nuevas bibliotecas, y en el refinamiento extensivo de las bibliotecas existentes. Estamos considerando la utilización de metodologías de diseño y *patterns* de hypermedia en el diseño de las mismas.

Es muy importante destacar que el desarrollo y refinamiento de las bibliotecas implica necesariamente su uso en aplicaciones reales. Por esto, otro punto importante es tomar *feedback* de grupos de desarrollo que utilicen HyCom.

11.3 Ambientes de Desarrollo

Una línea muy interesante involucra la realización de ambientes de desarrollo. Nuestro objetivo primario es el desarrollo de asistentes de autoría, que complementen

las bibliotecas de autoría a modo de *Wizards*.

Una posibilidad interesante es la de diseñar una herramienta CASE basada en metodologías de hypermedia, utilizando HyCom como el lenguaje subyacente para describir los diseños.

11.4 Álgebra de Combinadores

El desarrollo de un álgebra de combinadores involucra determinar formalmente las propiedades de los componentes y las combinaciones que pueden realizarse sobre los mismos. De esta forma, puede determinarse rigurosamente qué combinaciones son equivalentes, y bajo qué términos pueden combinarse ciertos componentes.

El desarrollo de un álgebra de combinadores es un paso fundamental si se desea considerar la posibilidad de desarrollar una herramienta CASE basada en HyCom. Esto permitiría validar ciertos aspectos importantes de la combinación de componentes y clarificar el significado de las combinaciones.

11.5 Validación

En conjunción con la verificación de ciertas propiedades formales, como lo propone el álgebra de combinadores, es posible utilizar ciertas heurísticas para la validación y medición de la calidad de los diseños.

Existen diferentes medidas heurísticas que facilitan la evaluación de aspectos que no pueden ser considerados de forma tan exacta como la estructura de las combinaciones (por ejemplo, factores humanos) [TM98]. Debido a que HyCom permite describir los diseños en un lenguaje expresivo y formal, aplicar estas heurísticas es mucho más efectivo que, por ejemplo, aplicarlas directamente a una aplicación implementada en HTML.

11.6 Integración con Tecnología de Componentes

Las aplicaciones que se mostraron en esta tesis usan CGI para manejar los aspectos dinámicos, utilizando CgiLib [vDM96]. Una alternativa sería estudiar el uso de la biblioteca CGI diseñada por John Hughes [Hug99]. Recientemente han aparecido diferentes tecnologías que permiten una mejor opción a la hora de implementar aplicaciones dinámicas. Estas tecnologías se basan generalmente en modelos de componentes como COM/DCOM [COM99, DCO99], JavaBeans [Jav99] o CORBA [COR99].

En el último año se han ido desarrollando bibliotecas que permiten la interacción de aplicaciones Haskell con COM/DCOM [MLH98]. Una línea de trabajo muy importante consiste en aplicar esta tecnología para el desarrollo de aplicaciones WWW con HyCom que vayan más allá del uso de CGI.

Capítulo 12

Conclusiones

En esta tesis presentamos HyCom, un lenguaje específico al dominio para el desarrollo de hypermedia. HyCom utiliza la metodología DSEL para proveer un marco en el cual desarrollar aplicaciones hypermediales con un alto nivel de abstracción.

La utilización de una metodología DSEL permite no sólo desarrollar un lenguaje tomando como principio las necesidades específicas del dominio, sino beneficiar al y beneficiarse del lenguaje huésped. Por un lado, HyCom aprovecha todos los DSELs ya existentes que utilizan el mismo huésped. Por otro lado, HyCom extiende al huésped, permitiendo desarrollar aplicaciones hypermediales en el mismo.

El lenguaje presentado basa su diseño en principios bien establecidos en la comunidad de hypermedia. Se proveen componentes reusables que permiten atacar un diseño en forma metodológica, separando las capas conceptuales, navegacionales y de interfaz. Esta estructura permite una integración directa con las metodologías de diseño existentes, ayudando a reducir el paso de diseño a implementación.

HyCom provee módulos específicos a plataformas de implementación, los cuales proveen funciones de traducción y componentes específicos a esa plataforma. Las funciones de traducción son una facilidad muy importante para permitir un enfoque por prototipos, el cual ha sido reconocido como fundamental en el desarrollo de este tipo de aplicaciones.

Concluímos esta tesis remarcando la originalidad de nuestro aporte. Los lenguajes declarativos clásicamente han sido dejados de lado en las aplicaciones no estrictamente académicas, como el desarrollo de hypermedia. Esta tesis demuestra que utilizar un lenguaje específico al dominio, embebido en un lenguaje declarativo, permite adoptar una forma de desarrollo de alto nivel de abstracción. El enfoque presentado permite además rápida integración con las metodologías de diseño existentes (conservando la riqueza conceptual), una facilidad no siempre encontrada en las herramientas comerciales. Vemos los resultados de esta tesis como un incentivo para la innovación en las técnicas de desarrollo de hypermedia, resultando en un claro beneficio para todas las áreas involucradas.

Parte V

Apéndices

Apéndice A

Introducción Rápida a Haskell

HyCom se basa en el paradigma de programación funcional (PF) y está embebido en el lenguaje funcional Haskell [PJH99]. Siendo un lenguaje embebido, HyCom hereda la rica estructura sintáctica y semántica de Haskell.

En este apéndice hacemos una revisión de las características básicas de PF y Haskell. El objetivo es introducir a los lectores a los conceptos básicos de PF que se requieren para comprender completamente el diseño de HyCom. Introducimos los conceptos de una manera informal, ignorando los detalles técnicos demasiado específicos. Comentamos también las características atractivas del paradigma. Los lectores con un interés más profundo en PF pueden remitirse a [BW88, Dav92, Bir98, Tho99, Hug89].

A.1 Conceptos Básicos

Un programa funcional está constituído íntegramente por expresiones y funciones. El valor de una expresión depende únicamente de las subexpresiones que lo conforman, no existiendo efectos laterales que puedan afectar el valor de la expresión; ésta es una característica fundamental del paradigma, denominada *transparencia referencial.*

Una función en PF es similar a una función en el sentido matemático. Por ejemplo, la siguiente es una función que computa una suma.

```
suma :: (Int, Int) -> Int
suma (x,y) = x + y
```

La primer ecuación del ejemplo es la *signatura de tipo* y dice que la función `suma` toma un par de valores de tipo `Int` y retorna otro `Int`. La segunda ecùación es la definición de la función y dice que la función asocia al par `(x,y)` el número resultante de sumar `x` y `y`.

El paradigma funcional tiene un objetivo importante: lograr un alto nivel de abstracción, de forma que el programador pueda razonar casi matemáticamente sobre la estructura de un programa, sin distraerse con detalles de implementación.

Esta característica es la que hace que en los programas funcionales las diferencias entre la especificación y la implementación sean muy reducidas.

La actual generación de lenguajes funcionales tiene sus orígenes en los fines de los años ochenta, siendo Haskell el estándar en el área. Las características más importantes de Haskell son la evaluación *lazy*, las funciones de alto orden, las funciones como tipos de datos, un sistema fuerte de tipos y estático con polimorfismo paramétrico, tipos de datos algebraicos y un sistema de clases para *overloading*.

Las funciones de alto orden son funciones que pueden tomar otras funciones como argumentos o que retornan funciones como su resultado. La evaluación parcial y la abstracción de las estructuras de control definidas por el usuario son algunas de las ventajas provistas.

Por ejemplo, la función `map` abstrae la estructura de control "aplicar una función a todos los valores de una lista". Una lista en Haskell se denota mediante corchetes (por ejemplo, `[1,2,3,4,5]` es una lista de enteros que aloja los valores del 1 al 5). Supongamos que deseamos aplicar la función `inc` (que toma un entero y lo incrementa) a todos los valores de la lista; hacemos esto con la expresión (`map inc [1,2,3,4,5]`), cuyo valor es la lista `[2,3,4,5,6]`. La función `map` es un ejemplo de función de alto orden; toma la función `inc` como argumento y retorna una función `map inc`, que toma una lista de números y retorna otra lista con todos los números incrementados.

Es importante remarcar el hecho de que al aplicar `map` a la función `inc` obtenemos otra función, a la que incluso podemos dar un nombre. Por ejemplo, la expresión `incList = map inc` define una función `incList` como la función retornada cuando `map` es aplicado a `inc`; la función `incList` toma la lista e incrementa todos sus componentes en uno. Hemos abstraído un patrón común (aplicar una función a todos los componentes de una lista) y podemos reutilizarlo para definir funciones específicas que se definen en base a este patrón (como `incList`).

Las funciones de alto orden también proveen una forma interesante de pensar sobre las funciones. Hemos visto previamente la función `suma` que toma un par de enteros y retorna otro entero. Una definición alternativa de `suma` (`suma'`) es la de una función que toma un entero y retorna otra función que toma un segundo entero y retorna el resultado de sumarlos.

```
suma' :: Int -> (Int -> Int)
suma' x y = x + y
```

La signatura de tipo dice que `suma'` toma un valor de tipo `Int` y retorna una función de tipo `(Int -> Int)`. Debido a que las signaturas de tipos son asociativas a derecha podemos eliminar los paréntesis; la signatura resultante puede escribirse como `sum :: Int -> Int -> Int`. La relación entre `suma` y `suma'` se denomina *currificación*. La ventaja de definir funciones mediante currificación radica en que la función puede tomar sus parámetros de a uno a la vez. De esta forma podemos reutilizar la función para definir otras funciones; por ejemplo, la función `inc` (que ya mencionamos) puede definirse utilizando `suma'` de la forma `inc = suma' 1`.

La *evaluación lazy* es un mecanismo de evaluación que sigue la regla "computar sólo cuando es necesario, y en tal caso, sólo una vez". La evaluación *lazy* mejora la modularidad, debido a que las funciones pueden programarse independientemente de otras, sin necesidad de pensar en el orden de evaluación, y aún así se obtiene eficiencia [Hug89]. Este es otro logro de la PF que contribuye a lograr un alto nivel de abstracción y razonamiento matemático. Por ejemplo, una porción de programa podría tener un *output* infinito, pero una función que la utilice puede necesitar sólo una parte, y entonces el programa termina a pesar de tener partes infinitas.

A.2 Tipos en Haskell

Haskell es un lenguaje fuertemente tipado. Su sistema de tipos es estático (el tipo de toda expresión es determinado en tiempo de compilación) y provee *inferencia de tipos* (los tipos pueden ser determinados por el compilador incluso aunque el programador no provea las signaturas correspondientes). Algunas de las características del sistema de tipos son el polimorfismo paramétrico, tipos algebraicos y clases de tipos para *overloading*.

El *polimorfismo paramétrico* es la capacidad de una función de operar sin tener conocimiento de algunos de sus parámetros. Esto permite, por ejemplo, tener código que trabaja sobre listas pero no conoce el tipo de los elementos de la lista. Un ejemplo típico es `map`; como `map` representa el patrón de "recorrer una lista aplicando una función" no necesita saber los tipos de los elementos de la lista. La signatura de tipos correspondiente a la función `map` es `map :: (a -> b) -> ([a] -> [b])`. La función `map` toma una función de valores de tipo `a` en valores de tipo `b` y retorna una función que toma una lista de valores de tipo `a`, devolviendo una lista de valores de tipo `b`; aquí `a` y `b` son *variables de tipo* y pueden instanciarse a cualquier tipo. Por lo tanto, las variables de tipo proveen un medio flexible para proveer polimorfismo paramétrico.

Los tipos definidos por el usuario son soportados en Haskell mediante los *sinónimos de tipos* y los *tipos algebraicos*. Un sinónimo de tipo le da un nuevo nombre a un tipo existente (no crea un tipo nuevo). Esto permite darle a los tipos nombres significativos en el contexto de una aplicación. Por ejemplo, el siguiente código define `Edad` como un nuevo nombre para un `Integer`.

```
type Edad = Integer
```

En Haskell podemos crear tipos nuevos utilizando tipos algebraicos. Estos proveen un mecanismo poderoso para definir tipos que combina flexibilidad con un alto nivel de abstracción. El usuario crea un nuevo tipo especificando su nombre y los objetos de datos permitidos para el mismo. Los tipos algebraicos son definidos mediante la palabra reservada `data`.

```
data Boolean = True | False
```

En la definición anterior, decimos que `Boolean` es un *constructor de tipos* y `True` y `False` son *constructores de datos*. Los constructores de datos determinan qué elementos están permitidos para el tipo. Ambos tipos de constructores pueden tener parámetros, que pueden ser elementos de otro tipo o incluso variables de tipos (permitiendo polimorfismo paramétrico, como ya vimos). El siguiente es un ejemplo de constructores parametrizados.

```
data Maybe a = Nothing | Just a
```

El tipo `Maybe a` se utiliza para que una función pueda devolver un valor de tipo `a` (`Just a`) o ningún valor `Nothing` en caso de error (por ejemplo, si el resultado de la computación está indefinido). Los tipos algebraicos también permiter expresar tipos recursivos, como puede verse en el siguiente ejemplo.

```
data BinTree a = Null | Node a (BinTree a) (BinTree a)
```

El ejemplo provee una definición recursiva de árbol binario que puede instanciarse para alojar elementos de cualquier tipo (esto es especificado haciéndolo un `BinTree` de objetos de tipo `a`, que es una variable de tipo). De esta manera, `BinTree Char` y `BinTree Boolean` son ejemplos de instanciaciones de `BinTree a` y representan árboles de caracteres y booleanos respectivamente. Observemos que el constructor de datos `Node` tiene tres parámetros: un valor de tipo `a` (el contenido del nodo) y dos subárboles de tipo `BinTree a` (que hacen a la definición recursiva del tipo).

El poder de los tipos algebraicos reside en su expresividad y nivel de abstracción. De los ejemplos mostrados podemos ver que los tipos de datos, incluso los recursivos, pueden ser expresados en forma sencilla sin necesidad de punteros u otra técnica de implementación de bajo nivel. Los elementos de datos se describen en forma simple, permitiendo al programador expresar claramente sus ideas sobre el modelo de datos de la aplicación.

A.3 Clases para Polimorfismo Ad-Hoc

Además del polimorfismo paramétrico, Haskell permite *polimorfismo ad-hoc* (*overloading*) [Jon95]. Este último es expresado en Haskell por medio de las *clases de tipos*. Una clase define un conjunto de operaciones y todos los tipos que son instancia de una clase deben proveer implementaciones para las mismas. Un claro ejemplo es la clase `Eq` (provista por las bibliotecas de Haskell); las instancias de la clase `Eq` son todos los tipos que proveen definiciones de los operadores `==` (comparación por igualdad) y `/=` (comparación por desigualdad).

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
   x /= y      = not (x==y)
```

La definición de Eq dice que las instancias de la clase deben proveer definiciones para las operaciones == y /= (observemos que podemos escribir también (==) y (/=), que son versiones prefijas de esos operadores infijos). También provee una definición por defecto para la operación /=, lo que significa que la definición para esta operación puede no proveerse si se acepta la implementación provista.

Las clases de tipos forman una jerarquía, en donde una subclase requiere todas las operaciones de la superclase y además sus propias operaciones. Por ejemplo, la clase Ord es una subclase de Eq y las instancias de Ord requieren proveer definiciones para la operación == (por ser instancias de Eq) y también operaciones de comparación como > ("mayor-que"), <= ("menor-o-igual-que"), que son especificadas por la clase Ord. Debemos aclarar que el hecho de que Ord sea subclase de Eq es una decisión de diseño de Haskell.

Las clases de tipos se utilizan en conjunción con los *contextos de clase*. Un contexto de clase permite establecer que determinada definición (de función o de tipo) es válida para todo elemento que sea instancia de una determinada clase. Por ejemplo, la siguiente definición establece que la función allEqual está definida para listas de elementos cuyo tipo sea instancia de Eq (o sea, para listas de elementos que puedan compararse por igualdad).

```
allEqual :: (Eq a) => [a] -> Bool
```

El contexto Eq a establece una restricción sobre la variable de tipo a. Los contextos pueden utilizarse también en una definición de tipos; el siguiente ejemplo define un árbol binario de búsqueda, restringiéndolo a los elementos cuyo tipo sea instancia de la clase Ord (o sea, elementos comparables).

```
data (Ord a) => BST a = NullBST | BST_Node a (BST a) (BST a)
```

Algunas clases de tipos tienen más de un parámetro, significando que el *overloading* se realiza sobre varios tipos a la vez, en lugar de uno solo. Estas clases se denominan *multi-parameter* [PJJM97] y representan un mecanismo de polimorfismo muy potente y frecuentemente no encontrado en otros lenguajes. Supongamos que queremos establecer la comparación por igualdad entre elementos de tipos diferentes (por ejemplo, para comparar enteros con reales); podemos definir una clase Comparable como sigue.

```
class Comparable a b where
   isEqual :: a -> b -> Bool
```

La definición de la operación isEqual deberá darse por cada par de parámetros a y b y el compilador decidirá que definición utilizar de isEqual en base a ambos tipos. Por ejemplo, podemos instanciar Comparable para poder comparar los números enteros con los reales.

```
instance Comparable Int Real where
   isEqual i r = (not (hasDecimals r)) &&
                 (round r == i)
```

Podemos definir otra instanciación para comparar números enteros con valores booleanos (claramente, la comparación siempre resultará falsa).

```
instance Comparable Int Bool where
   isEqual _ _ = False
```

La elección de la instancia particular de `isEqual` depende de los tipos de los parámetros, con lo cual `isEqual 1 1.0` y `isEqual 2 True` utilizarán diferentes implementaciones de `isEqual` para realizar la computación. Aclaramos que en algunos casos es necesario aclarar los tipos de los parámetros en forma explícita para la correcta resolución del `overloading`.

Concluyendo, las clases de tipos son una técnica poderosa para el modelado de aplicaciones en Haskell y son muy utilizadas en el trabajo presentado en esta tesis.

A.4 Flexibilizando el Tipado Fuerte

Uno de los puntos más discutidos en todo lenguaje de programación radica en las ventajas y desventajas del tipado fuerte. Si bien no podemos ignorar las ventajas del chequeo estático, es claro que el chequeo dinámico de tipos otorga gran flexibilidad a la hora del desarrollo de aplicaciones reales. El lenguaje Haskell provee un interesante mecanismo que permite al tipado fuerte acercarce a la flexibilidad del tipado dinámico, sin perder las ventajas del primero. Veremos el funcionamiento de este mecanismo con un ejemplo práctico.

Una de las características más atractivas de los lenguajes con chequeo dinámico (por ejemplo, Smalltalk [GR83]) radica en la posibilidad de tener colecciones heterogéneas. En un principio, la aproximación más cercana que podríamos ofrecer en Haskell consistiría en restringir los posibles tipos a alojar en la colección y agruparlos en un tipo `Element`.

```
data Element = An_Integer Integer |
               A_Char Char        |
               A_Bool Bool

type Collection = [Element]
```

El tipo `Collection` permite almacenar elementos de los tipos `Integer`, `Char` y `Bool`, encapsulados dentro de un tipo `Element`. Claramente esto atenta contra la flexibilidad, ya que si necesitamos alojar en la colección un tipo nuevo, necesitamos extender el tipo `Element`.

En el caso anterior, el tipo `Element` describía *por extensión* el conjunto de elementos que podía alojar en su interior. Consideremos qué sucede si en vez de describir el conjunto por extensión lo hacemos *por comprensión*; es decir, en vez de enumerar los elementos, describimos la propiedad que deben cumplir. En Haskell podemos expresar una propiedad de un conjunto de tipos mediante una clase (por ejemplo, la clase `Show` representa la propiedad de poder convertirse a `String`).

```
class Show a where
   show :: a -> String
   ...
```

Habiendo visto cómo representar una propiedad con una clase, deseamos ahora poder definir un tipo capaz de alojar a cualquier elemento de esa clase. De esta forma habremos definido un tipo que puede almacenar a un conjunto de elementos descrito por una propiedad determinada. Podemos hacer esto en Haskell mediante un mecanismo llamado *cuantificación* [Jon99]; el siguiente ejemplo utiliza el operador de cuantificación (`forall`) para expresar que el tipo `AnyElement` puede alojar elementos de cualquier tipo que sea instancia de la clase `Show`.

```
data AnyElement = forall a. Show a => Any_Element a
```

Como puede verse, `AnyElement` no tiene ningún parámetro que establezca qué tipo contiene. En cambio, sólo se sabe que los elementos que contiene cumplen con las propiedades especificadas por `Show` (es decir, proveen la función `show` entre otras). Podemos ahora definir una nueva versión de colección utilizando `AnyElement`.

```
type NewCollection = [AnyElement]
```

Esta colección es mucho más flexible que la versión anterior, puesto que puede contener cualquier tipo de elemento cuyo tipo sea instancia de `Show`. Hemos restringido el concepto de colección heterogénea, permitiendo sólo alojar elementos que cumplen cierta propiedad; aún así logramos más flexibilidad que con la enumeración de los tipos permitidos (con esta nueva versión, sólo es necesario instanciar un tipo con la clase apropiada para almacenarlo en la colección). Claramente, las únicas operaciones permitidas sobre los elementos son aquellas provistas por la clase `Show`; por ejemplo, la siguiente es una manipulación válida de la colección.

```
getStrings :: NewCollection -> [String]
getStrings =  map toString
   where toString (Any_Element a) = show a
```

La función `getStrings` utiliza una función `toString`, que extrae el elemento y le aplica `show`. El tipo del elemento es desconocido y sólo se sabe que pertenece a la clase `Show`; aplicar una operación no perteneciente a `Show` es inválido (ya que el chequeador no puede verificar que los tipos concuerden correctamente). De la misma forma, es imposible extraer el elemento contenido dentro de `AnyElement`, debido a que se desconoce su tipo.

```
extract :: AnyElement -> ???
extract (Any_Element a) = a
```

En definitiva, el mecanismo de cuantificación permite realizar construcciones similares a las obtenidas con el tipado dinámico (con algunas restricciones), sin

perder las ventajas del tipado fuerte. En particular, la cuantificación nos permite incorporar una versión restringida de colecciones heterogéneas (este tipo de colecciones son utilizadas reiteradamente en HyCom).

Debemos aclarar que el mecanismo fue explicado en forma breve e informal, a fin de no entrar en detalles técnicos que dificulten la lectura (el lector interesado puede consultar las referencias). Aclaramos también que la forma de cuantificación ilustrada aquí también es denominada *cuantificación existencial* (a pesar de que el operador se llame `forall`) por motivos históricos.

A.5 Entrada/Salida Monádica

La PF obtiene gran parte de su claridad y estilo matemático del hecho de evitar los efectos laterales en los programas. Sin embargo, cualquier aplicación real necesita de los efectos laterales para comunicarse con el mundo exterior mediante entrada/salida (E/S). Los diferentes lenguajes funcionales resuelven el problema de la E/S de diferentes formas; mientras que algunos optan por resignar su pureza funcional permitiendo efectos laterales [Aug84], otros (como Haskell) utilizan modelos de E/S puramente funcionales (que preservan la transparencia referencial).

Existen varios modelos de E/S puramente funcional [HS88, HC95], siendo el más popular y usado el modelo *monádico* [Wad95, GH95]. Las mónadas son un concepto proveniente de la teoría de categorías, y fueron muy utilizadas para expresar la semántica operacional de los lenguajes imperativos. No vamos a entrar en detalles técnicos o teóricos sobre los fundamentos de las mónadas, sino que nos limitaremos a mostrar cómo se utilizan para realizar E/S en Haskell.

En Haskell la mónada encargada de la E/S está representada por el tipo `IO a` (donde `a` es una variable de tipo). Básicamente, si una función devuelve un tipo `IO a`, esto significa que la función realiza algún efecto lateral y el resultado de ese efecto es de tipo `a`. Por ejemplo, la función `readFile` lee una secuencia de caracteres de un archivo, y tiene la siguiente signatura de tipo.

```
readFile :: FileName -> IO Char
```

No necesariamente el resultado del efecto lateral debe ser un elemento concreto. Por ejemplo, la función `writeFile` sólo realiza un efecto lateral (escribir un `String` en un archivo) sin obtener ningún elemento como resultado del efecto (por lo tanto, su resultado es de tipo `IO()`).

```
writeFile :: FileName -> String -> IO()
```

El resultado de evaluar funciones como las anteriores no es el resultado de su efecto lateral, sino que es la acción misma. Por ejemplo, si evaluamos la expresión `readFile "Read.Me"` (donde `Read.Me` es un archivo de texto) en un intérprete Haskell, obtendremos que el valor de la evaluación es `<<action>>`. Las funciones monádicas devuelven acciones que pueden ejecutarse y secuenciarse en condiciones

controladas, de forma de no alterar la transparencia referencial; en Haskell las mónadas se utilizan normalmente en un bloque `do`, que permite secuenciar las acciones con un estilo similar al de los lenguajes imperativos.

```
example :: IO()
example  = do
             temp <- readFile "In.Txt"
             writeFile "Out.Txt" temp
```

La función anterior devuelve una acción de tipo `IO()`; la misma es el resultado de ejecutar `readFile "In.Txt"` (el resultado del efecto lateral puede asignarse a una variable temporal `temp`), y luego ejecutar `writeFile ...`, con lo cual se escribe lo leído a un archivo. El estilo de la declaración es claramente imperativo, pero la transparencia referencial se preserva.

El repaso que realizamos de la entrada/salida monádica está sobresimplificado intencionalmente. Explicamos únicamente los conceptos necesarios para comprender su uso en esta tesis, dejando al lector la opción de consultar la bibliografía relacionada [GH95].

A.6 Resumen

En este apéndice repasamos algunas de las características de la PF y Haskell. En cada caso nos limitamos a comentar informalmente los puntos necesarios para el entendimiento correcto de esta tesis y recomendamos que el lector interesado se remita a la bibliografía citada. Para concluir el apéndice creemos conveniente remarcar algunas de los puntos que hacen interesante al paradigma.

Por un lado, la PF es atractiva debido a que alcanza un alto grado de modularidad y permite el reuso en gran escala. Esto se da principalmente porque las funciones de alto orden nos permiten abstraer patrones usados comúnmente (como `map`) y reutilizarlos.

Por otro lado, la naturaleza matemática de la PF es muy adecuada para expresar especificaciones y diseños, que pueden escribirse casi directamente en Haskell u otro lenguaje funcional (obteniéndose así prototipos ejecutables). Consecuentemente, la distancia entre la especificación, el diseño y la implementación se reduce, ya que el mismo formalismo es utilizado para las tres actividades.

Apéndice B

Referencia Técnica de HyCom

En este apéndice se da un sumario de las clases presentes en HyCom, especificando cuál es el significado de cada una y el conjunto de operaciones que definen.

B.1 Modelo Básico

- **Clase** `Component`

  Esta clase modeliza la funcionalidad básica de un componente dentro de HyCom.

  Operaciones

  - `getID :: a -> ID`
    Retorna el `ID` de un componente de tipo `a`, siendo `a` un tipo instancia de la clase.
  - `setID :: ID -> a -> a`
    Setea el `ID` de un componente de tipo `a`, siendo `a` un tipo instancia de la clase.

  Instancias

  - `(Component a, Component b) => Combined c a b`
    Modeliza la forma más básica de combinación.
  - `(Component a) => Transformed t a`
    Modeliza la forma más básica de transformación.

B.2 Modelo Navegacional

- **Clase** `NodeComponent`

  Representa un nodo en su forma más básica. Un nodo es un componente que posee un cierto conjunto de *links* que parten de él.

Superclases

- `Component`

Operaciones

- `getOutgoingLinks :: a -> Links`
  Retorna los *links* que parten del nodo.
- `getNeighbourhood :: a -> Nodes`
  Retorna los nodos adyacentes al nodo tomado como parámetro.

- Clase `Hyperlink`

Representa un *link* en su forma más básica. Un *link* es un elemento de asociación entre componentes que representan nodos. Un *link* conoce el identificador de su nodo origen y el de su nodo destino.

Operaciones

- `getOriginNode :: a -> ID`
  Retorna el `ID` del nodo origen del *link*.
- `getDestinyNode :: a -> ID`
  Retorna el `ID` del nodo destino del *link*.

Instancias

- `(NodeComponent o, NodeComponent d) => PlainLink o d`
  Modeliza la forma más simple de *link*. El tipo del *link* depende de los tipos de los nodos de origen y destino.
- `SimpleLink`
  Modeliza la forma más simple de *link*. A diferencia de `PlainLink`, el tipo de `SimpleLink` no depende de los tipos del origen y del destino.

- Clase `HasInternalDestiny`

Representa una forma especializada de *link*, donde el destino es un componente específico dentro de un nodo particular.

Superclases

- `Hyperlink`

Operaciones

- `getInternalDestiny :: a -> ID`
  Retorna el componente de destino que se encuentra dentro del nodo destino del *link*.

- Clase `HasInternalOrigin`

  Representa una forma especializada de *link*, donde el origen es un componente específico dentro de un nodo particular.

  Superclases

  - `Hyperlink`

  Operaciones

  - `getInternalOrigin :: a -> ID`
    Retorna el componente de origen que se encuentra dentro del nodo origen del *link*.

- Clase `Context`

  Representa un contexto. Un contexto es un componente que agrupa varios nodos de acuerdo a una política particular. Un contexto tiene además una política de transformación de identificadores, de forma que todos los nodos presentes en un contexto ven modificado su identificador de la misma manera. La transformación de identificadores permite diferenciar el mismo nodo en diferentes contextos y realizar *hyperlinking* en forma acorde.

  Superclases

  - `Component`

  Operaciones

  - `inContextNodeID :: c -> ID -> ID`
    Dado el `ID` de un nodo y un contexto, transforma el mismo para que represente el `ID` de un nodo dentro del contexto. Todos los `ID` de los nodos dentro de un contexto se transforman mediante esta función.
  - `getNodesInContext :: c -> [ID]`
    Retorna los `ID` de los nodos dentro del contexto.
  - `inContextNode :: (NodeComponent n) => c -> n -> n`
    Dado un nodo, transforma su `ID` utilizando la función `inContextNodeID`.
  - `getContextDescription :: c -> String`
    Retorna un `String` describiendo al contexto.

- Clase `IndexedContext`

  Representa un contexto indexado. Un contexto indexado es aquel que contiene un índice de los nodos que lo forman, permitiendo el acceso a los mismos a través de él.

  Superclases

- `Context`
- `Component`

Operaciones

- `contextIndex :: c -> ContextIndex s`
  Retorna el índice del contexto, de tipo `(ContextIndex s)`. Un índice de ese tipo mantiene el índice del contexto formado por los `IDs` de los nodos y por las claves de tipo `s`.

- **Clase `SequentialContext`**

Representa un contexto secuencial. Un contexto secuencial determina que los nodos se encuentran ordenados secuencialmente, de manera tal que para cada nodo se tiene un nodo anterior y un nodo siguiente. La política de nodo anterior y nodo siguiente es determinada internamente por el componente que implementa el contexto.

Superclases

- `Context`
- `Component`

Operaciones

- `nextInContext ::`
  `(NodeComponent n) => c -> n -> Maybe ID`
  Dado un contexto y un nodo retorna el `ID` del siguiente nodo en el contexto si existe.
- `previousInContext ::`
  `(NodeComponent n) => c -> n -> Maybe ID`
  Dado un contexto y un nodo retorna el `ID` del nodo previo en el contexto si existe.
- `toNextInContext ::`
  `(NodeComponent n) => c -> n -> Maybe SimpleLink`
  Dado un contexto y un nodo retorna un *link* al siguiente nodo en el contexto si existe.
- `toPreviousInContext ::`
  `(NodeComponent n) => c -> n -> Maybe SimpleLink`
  Dado un contexto y un nodo, retorna un *link* al nodo previo en el contexto si existe.

- Clase `InContext`

  Esta clase permite representar la propiedad de que un nodo pertenece a un contexto particular. Permite acceder a las características del contexto a partir de un nodo.

  Superclases

  - `NodeComponent`
  - `Component`

  Operaciones

  - `nodeGetContextDescription :: ID -> n -> String`
    Retorna la descripción del contexto con identificador `ID`, en el cual se encuentra el nodo.
  - `getContexts :: n -> [ID]`
    Retorna los `ID` de todos los contextos en los cuales se encuentra el nodo.

- Clase `InIndexedContext`

  Esta clase permite representar la propiedad de que un nodo pertenece a un contexto indexado particular. Permite recuperar el índice del contexto a partir de un nodo.

  Superclases

  - `NodeComponent`
  - `Component`

  Operaciones

  - `nodeContextIndex :: ID -> n -> ContextIndex s`
    Retorna el índice del contexto con identificador `ID`, en el cual se encuentra el nodo.

- Clase `InSequentialContext`

  Esta clase permite representar la propiedad de que un nodo pertenece a un contexto secuencial particular. Permite acceder a la funcionalidad del contexto secuencial a partir de un nodo.

  Superclases

  - `NodeComponent`
  - `Component`

  Operaciones

- `nodeNextInContext :: ID -> n -> Maybe ID`
  Retorna el identificador del siguiente nodo en el contexto de identificador `ID`, en el cual se encuentra el nodo.
- `nodePreviousInContext :: ID -> n -> Maybe ID`
  Retorna el identificador del nodo previo en el contexto de identificador `ID`, en el cual se encuentra el nodo.
- `nodeToNextInContext :: ID -> n -> Maybe SimpleLink`
  Retorna un *link* al siguiente nodo en el contexto de identificador `ID`, en el cual se encuentra el nodo.
- `nodeToPreviousInContext ::`
  `ID -> n -> Maybe SimpleLink`
  Retorna un *link* al nodo previo en el contexto de identificador `ID`, en el cual se encuentra el nodo.

B.3 Interfaz del Usuario

- **Clase `UIComponent`**

  Representa un componente de interfaz del usuario en su forma más básica, determinando los atributos presentes en todo componente de interfaz. Un componente de esta clase provee colores de frente y de fondo.

  Superclases

  - `Component`

  Operaciones

  - `getBackground :: a -> Color`
    Retorna el color de fondo del componente.
  - `getForeground :: a -> Color`
    Retorna el color de frente del componente.
  - `setBackground :: Color -> a -> a`
    Setea el color de fondo del componente.
  - `setForeground :: Color -> a -> a`
    Retorna el color de frente del componente.

- **Clase `HasUI`**

  Esta clase permite definir la propiedad de un componente de tener una representación de interfaz asociada. Un componente que es instancia de esta clase puede determinar cuál es el identificador del componente de interfaz que lo representa.

  Superclases

- `Component`

Operaciones

- `getUI :: a -> ID`
  Retorna el `ID` de la interfaz del componente.

- **Clase `HyperlinkUI`**

  Esta clase permite definir la propiedad de un *link* de poder determinar cuáles son las representaciones de interfaz de sus nodos origen y destino. La clase permite determinar la representación de una relación de navegación en la interfaz.

  Superclases

  - `Hyperlink`

  Operaciones

  - `getOriginNodeUI :: a -> ID`
    Retorna el `ID` de la interfaz del nodo origen.
  - `getDestinyNodeUI :: a -> ID`
    Retorna el `ID` de la interfaz del nodo destino.

- **Clase `HasInternalDestinyUI`**

  Esta clase es análoga a `HyperlinkUI`, pero para *links* con un destino interno a un nodo.

  Superclases

  - `HyperlinkUI`

  Operaciones

  - `getInternalDestinyUI :: a -> ID`
    Retorna el `ID` de la interfaz del componente destino, que se encuentra dentro de la interfaz del nodo destino.

- **Clase `HasInternalOriginUI`**

  Esta clase es análoga a `HyperlinkUI`, pero para *links* con un origen interno a un nodo.

  Superclases

  - `HyperlinkUI`

Operaciones

- `getInternalOriginUI :: a -> ID`
  Retorna el `ID` de la interfaz del componente origen, que se encuentra dentro de la interfaz del nodo origen.

- **Clase `SpecificationUI`**

  Esta clase representa una especificación de un conjunto de componentes de UI relacionados.

  Operaciones

  - `getUIComponents :: s -> UIComponents`
    Retorna el conjunto de componentes de interfaz que constituyen la especificación.
  - `areConnectedUI ::`
    `s -> AnUIComponent -> AnUIComponent -> Links`
    Retorna los *links* que unen a los componentes de interfaz tomados como parámetro.

  Instancias

  - `(UIComponent a) => BasicSpecificationUI a`
    Es el tipo más básico de especificación de interfaz del usuario.

- **Clase `HasImage`**

  Representa un componente de interfaz que posee una imagen.

  Superclases

  - `UIComponent`
  - `Component`

  Operaciones

  - `getSource :: a -> Source`
    Retorna la fuente de la imagen.
  - `getDescription :: a -> Description`
    Retorna la descripción de la imagen.
  - `setSource :: Source -> a -> a`
    Setea la fuente de la imagen.
  - `setDescription :: Description -> a -> a`
    Setea la descripción de la imagen.

- Clase `HasText`

  Representa un componente de interfaz que posee texto.

  Superclases

  - `UIComponent`
  - `Component`

  Operaciones

  - `getText :: a -> String`
    Retorna el texto del componente.
  - `setText :: String -> a -> a`
    Setea el texto del componente.

- Clase `HasLength`

  Representa un componente de interfaz que posee texto y además posee restricciones sobre la longitud del texto permitido.

  Superclases

  - `HasText`
  - `UIComponent`
  - `Component`

  Operaciones

  - `getLength :: a -> Length`
    Retorna la longitud de texto del componente.
  - `setLength :: Length -> a -> a`
    Setea la longitud de texto del componente.

- Clase `HasStyle`

  Representa un componente de interfaz que posee texto y además posee un atributo especificando el estilo del texto.

  Superclases

  - `HasText`
  - `UIComponent`
  - `Component`

  Operaciones

- `getStyle :: a -> TextStyle`
  Retorna el estilo de texto del componente.
- `setStyle :: TextStyle -> a -> a`
  Setea el estilo de texto del componente.

- **Clase `HasFont`**

  Representa un componente de interfaz que posee texto y además posee un atributo especificando el tipo de letra (*font*) del texto.

  Superclases

  - `HasText`
  - `UIComponent`
  - `Component`

  Operaciones

  - `getFont :: a -> TextFont`
    Retorna el *font* del texto del componente.
  - `setFont :: TextFont -> a -> a`
    Setea el *font* del texto del componente.

- **Clase `HasTextSize`**

  Representa un componente de interfaz que posee texto y además posee un atributo especificando el tamaño de letra usado en el texto.

  Superclases

  - `HasText`
  - `UIComponent`
  - `Component`

  Operaciones

  - `getTextSize :: a -> TextSize`
    Retorna el tamaño del texto del componente.
  - `setTextSize :: TextSize -> a -> a`
    Setea el tamaño del texto del componente.

- **Clase `HasDimensions`**

  Representa un componente de interfaz que posee texto y además posee dimensiones que restringen el tamaño del área de texto permitida.

  Superclases

- `HasText`
- `UIComponent`
- `Component`

Operaciones

- `getRows :: a -> Rows`
  Retorna la cantidad de filas del área de texto del componente.
- `getColumns :: a -> Columns`
  Retorna la cantidad de columnas del área de texto del componente.
- `setRows :: Rows -> a -> a`
  Setea la cantidad de filas del área de texto del componente.
- `setColumns :: Columns -> a -> a`
  Setea la cantidad de columnas del área de texto del componente.

- **Clase `HasSize`**

Representa un componente de interfaz con una indicación de tamaño. La misma no es necesariamente el tamaño del componente cuando se lo visualiza, sino que cuantifica algún atributo del mismo (por ejemplo, número de entradas en una lista).

Superclases

- `UIComponent`
- `Component`

Operaciones

- `getSize :: a -> Size`
  Retorna la indicación de tamaño del componente.
- `setSize :: Size -> a -> a`
  Setea la indicación de tamaño del componente.

- **Clase `Groupable`**

Representa un componente de interfaz que puede agruparse con otros componentes, resultando en un comportamiento conjunto particular. El ejemplo típico son los *radio buttons*.

Superclases

- `UIComponent`
- `Component`

Operaciones

- `getGroupID :: a -> GroupID`
  Retorna el `ID` del grupo del componente.
- `setGroupID :: GroupID -> a -> a`
  Setea el `ID` del grupo del componente.

- Clase `HasAction`

  Representa un componente de interfaz que tiene una acción asociada. Un ejemplo sencillo es un botón, que ejecuta algún tipo de acción al ser presionado. Esta clase es una *multi-parameter*; llamaremos a sus parámetros `a` y `action`.

  ### Superclases

  - `UIComponent`
  - `Component`

  ### Operaciones

  - `getAction :: c -> action`
    Retorna la acción del componente.
  - `setAction :: action -> c -> c`
    Setea la acción del componente.

- Clase `Checkable`

  Representa un componente de interfaz con un atributo que indica si el mismo se encuentra en el estado chequeado o no-chequeado. El ejemplo típico son los *check boxes*.

  ### Superclases

  - `UIComponent`
  - `Component`

  ### Operaciones

  - `getChecked :: a -> Checked`
    Retorna el estado de chequeo del componente.
  - `setChecked :: Checked -> a -> a`
    Setea el estado de chequeo del componente.

- **Clase `Selectable`**

  Representa un componente de interfaz que permite realizar una selección entre un conjunto de opciones posibles. El ejemplo típico lo constituyen los *group boxes* y los *list panes*.

  Superclases

  - `UIComponent`
  - `Component`

  Operaciones

  - `getOptions :: a -> [Option]`
    Retorna las opciones del componente.
  - `setOptions :: [Option] -> a -> a`
    Setea las opciones del componente.

- **Clase `Multiple`**

  Representa un componente de interfaz que permite seleccionar entre un conjunto de opciones posibles y tiene un atributo que indica si es posible o no seleccionar más de una opción simultáneamente.

  Superclases

  - `Selectable`
  - `UIComponent`
  - `Component`

  Operaciones

  - `getMultiple :: a -> Bool`
    Retorna una indicación de si el componente permite selección múltiple o no.
  - `setMultiple :: Bool -> a -> a`
    Setea una indicación de si el componente permite selección múltiple o no.

- **Clase `HasState`**

  Representa los componentes que tienen un estado interno. Esta clase es una *multi-parameter*; llamaremos a sus parámetros `c` y `state`.

  Operaciones

  - `getState :: c -> state`
    Retorna el estado del componente.
  - `setState :: state -> c -> c`
    Setea el estado del componente.

B.4 Utilidades Generales

- Clase `Retriever`

  Esta clase modeliza la funcionalidad de poder recuperar cierto tipo de componente de algún medio de almacenamiento determinado. Esta clase es una *multi-parameter*; llamaremos a sus parámetros `media` y `component`.

  Operaciones

  - `retrieve :: media -> IO [component]`
    Recupera todos los componentes de tipo `component` en el medio de almacenamiento indicado.
  - `retrieveBy ::`
    `media -> (component -> Bool) -> IO [component]`
    Recupera los componentes de tipo `component` en el medio de almacenamiento indicado que satisfacen el predicado especificado.

  Instancias

  - `(Read a) => (TrivialRetriever a) a`
    Es la forma más simple de *retriever*, y recupera los componentes de un archivo de texto. Está disponible automáticamente para todo componente que pueda ser parseado desde un *string*.

- Clase `Storer`

  Esta clase modeliza la funcionalidad de poder almacenar cierto tipo de componente en un cierto medio de almacenamiento. Esta clase es una *multi-parameter*; llamaremos a sus parámetros `media` y `component`.

  Operaciones

  - `store :: media -> [component] -> IO()`
    Almacena la lista de componentes en el medio de almacenamiento indicado.

  Instancias

  - `(Show a) => (TrivialStorer a) a`
    Es la forma más simple de *storer* y almacena los componentes en un archivo de texto. Está disponible automáticamente para todo componente que pueda ser transformado a un *string*.

Apéndice C

Aplicaciones realizadas con HyCom

En este apéndice mostramos las aplicaciones reales que fueron realizadas mediante la primera versión de HyCom (presentada en [MMLR97] y [MMLR98]). El desarrollo de estas aplicaciones nos permitió incrementar nuestra experiencia en el dominio; esto nos resultó de gran utilidad en el diseño de la versión de HyCom presentada en esta tesis.

C.1 Páginas del CLaPF97

En 1997 estuvimos encargados de desarrollar el *site* de la *Segunda Conferencia Latinoamericana de Programación Funcional (CLaPF97)*. Esta representó la primer aplicación real de HyCom y nos reportó interesantes experiencias.

El *site* de *CLaPF97* tenía una estructura de información bien definida, haciéndolo muy apto para modelarlo en HyCom. Básicamente, permitía acceder a información sobre la organización del congreso, los trabajos presentados y las charlas de investigadores invitados.

Algunas páginas de este *site* se muestran en las Figs. C.1 y C.2.

C.2 Páginas Personales

En 1998 realizamos nuestras páginas personales mediante HyCom. Utilizar el DSEL nos permitió definir la estructura de información de las páginas en forma separada de la presentación, con lo cual pudimos instanciar nuestras páginas a partir de una estructura general de página personal. Esta representó una primera aproximación al concepto de biblioteca de autoría (ver Cap. 7).

Las páginas desarrolladas contienen información personal y sobre los *papers* publicados. Mostramos las páginas en las Figs. C.3, C.4 y C.5.

C.3 Páginas del HyCom

A fin de promover el uso de HyCom y obtener *feedback* de los usuarios, definimos una página de soporte al DSEL (ver Fig. C.6). En la misma, los usuarios pueden obtener el paquete completo de HyCom, los *papers* publicados y un *tutorial*. Aclaramos que la página cubre la primera versión del DSEL y no la mostrada en esta tesis.

C.4 Páginas del Tutorial del HyCom

Con motivos de facilitar el aprendizaje de HyCom por parte de los usuarios desarrollamos un *tutorial* hypermedial. El *tutorial* está organizado en secciones, describiendo los diferentes aspectos del DSEL. Algunas páginas del *tutorial* pueden verse en las Figs. C.7, C.8, C.9 y C.7.

C.5 Páginas del Grupo BioCom

El grupo BioCom realizó actividades de investigación dentro del LIFIA hasta 1998, siendo su área de trabajo la Biología Computacional. El grupo necesitaba mantener un *site* bien organizado que mantuviera la información sobre sus actividades y publicaciones. Decidimos modelar el *site* de BioCom utilizando HyCom, lo cual nos permitió organizar efectivamente la información contenida dentro del *site* y uniformizar su presentación visual. Algunas páginas de este *site* se muestran en las Figs. C.11 y C.12.

C.6 Páginas sobre la Tecnología de Agentes

A fines de 1998 se realizaron varias reuniones informales sobre el tema agentes en nuestro grupo de trabajo. Disponíamos entonces de muchas referencias a información y necesitábamos organizarla y permitir su acceso a través de la WWW. Construímos un *site* a partir de esa información utilizando HyCom. Algunas páginas del *site* se muestran en las Figs. C.13 y la C.14.

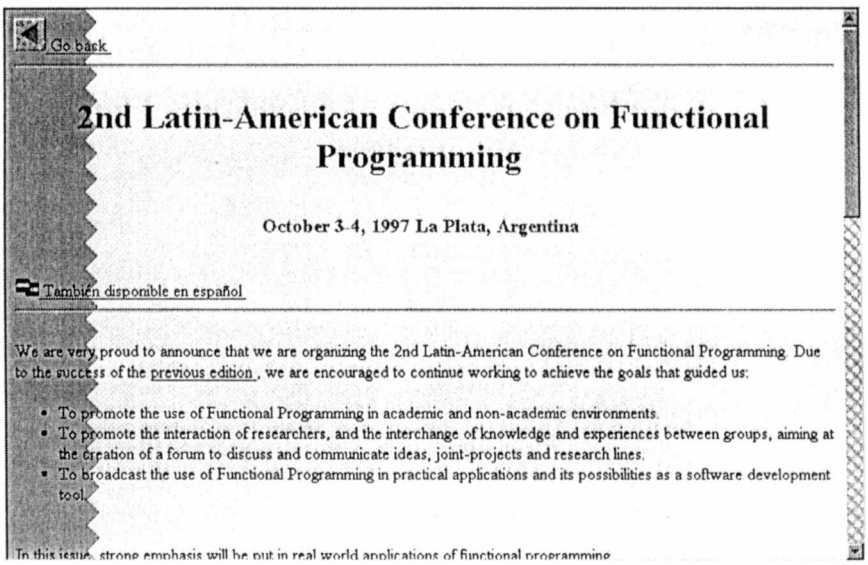

Figura C.1: *Home Page* del CLaPF97

Figura C.2: Charlas Invitadas del CLaPF97

Figura C.3: *Home Page* de Daniel H. Marcos

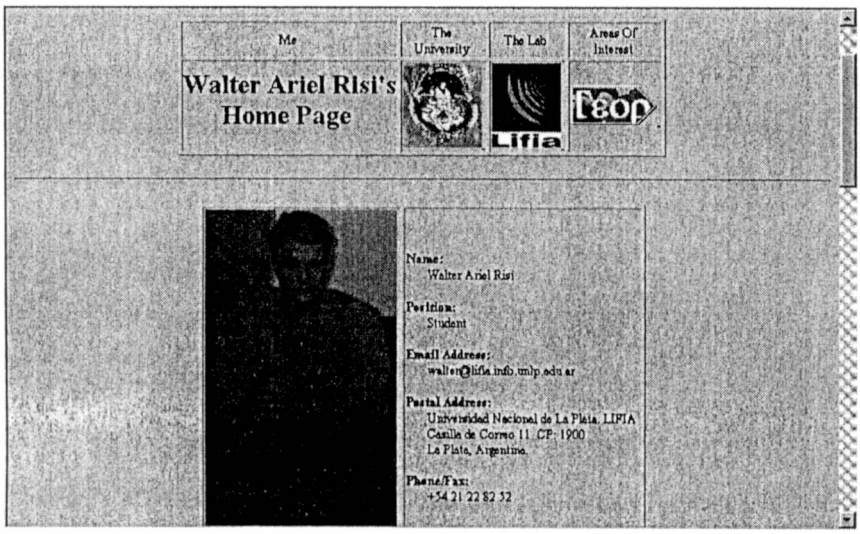

Figura C.4: *Home Page* de Walter A. Risi

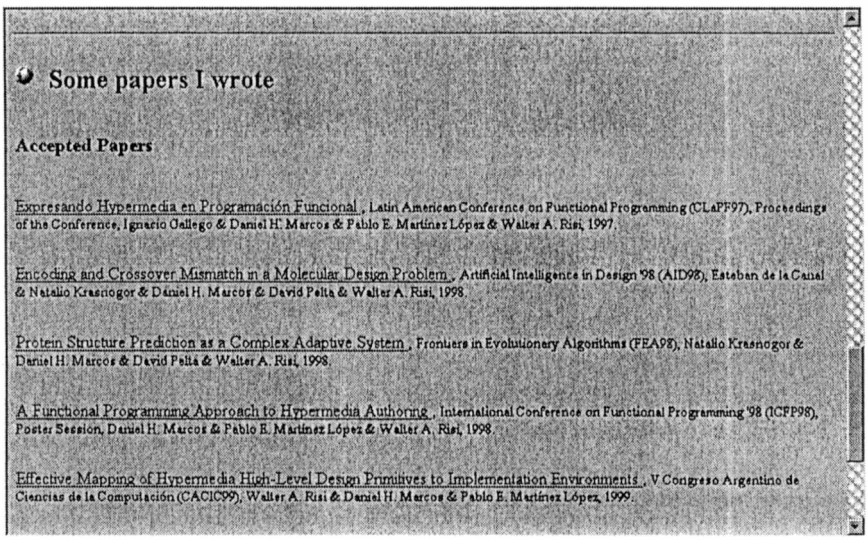

Figura C.5: *Papers* Publicados

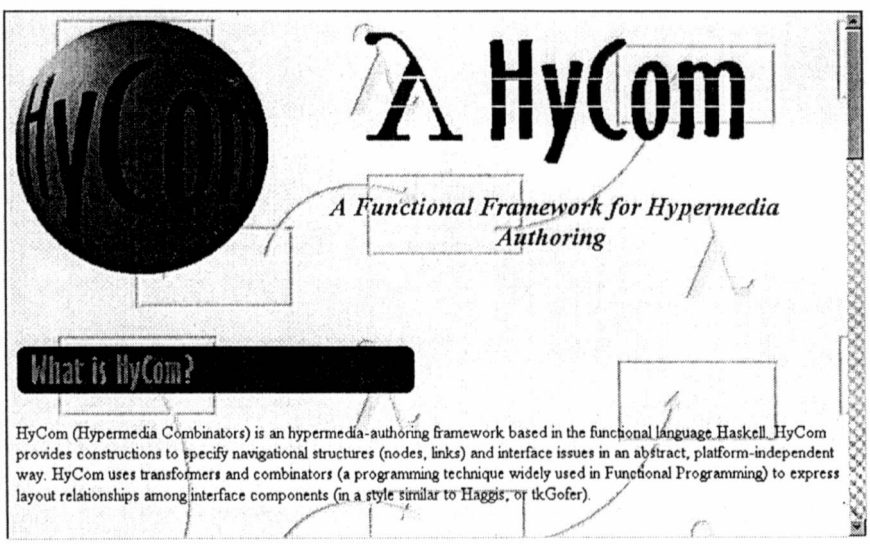

Figura C.6: *Home Page* de HyCom

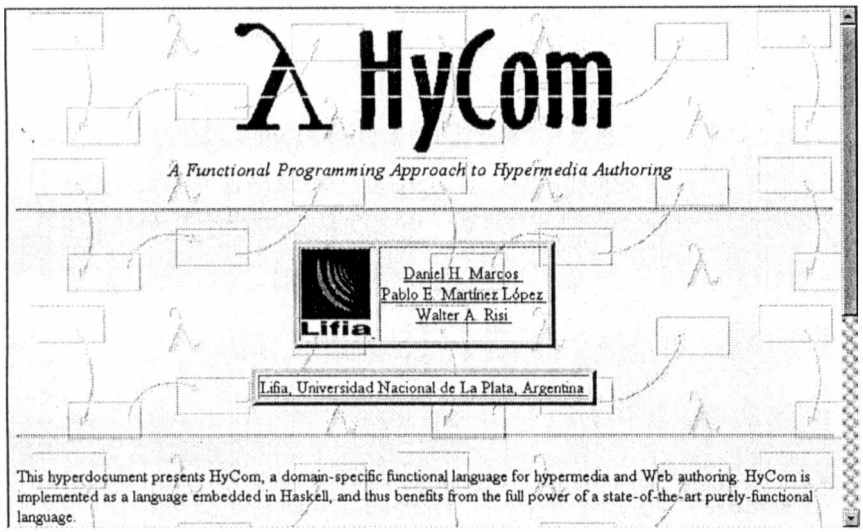

Figura C.7: *Home Page* del *Tutorial* del HyCom 1.0

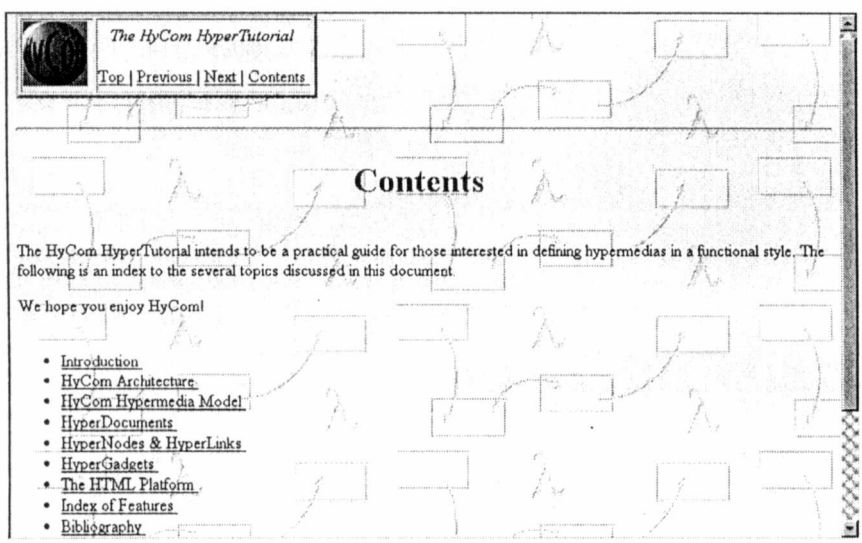

Figura C.8: Página del Contenido del *Tutorial* del HyCom 1.0

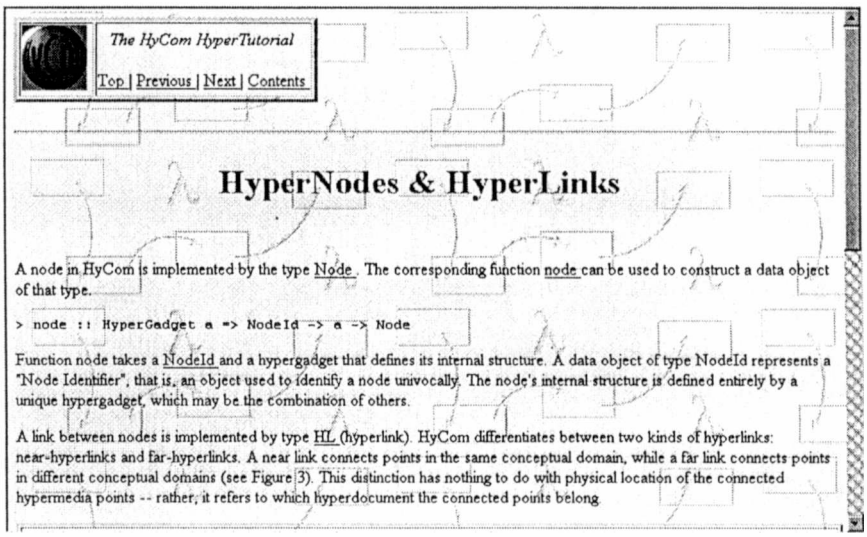

Figura C.9: Página Explicativa de los HyperNodes del HyCom 1.0

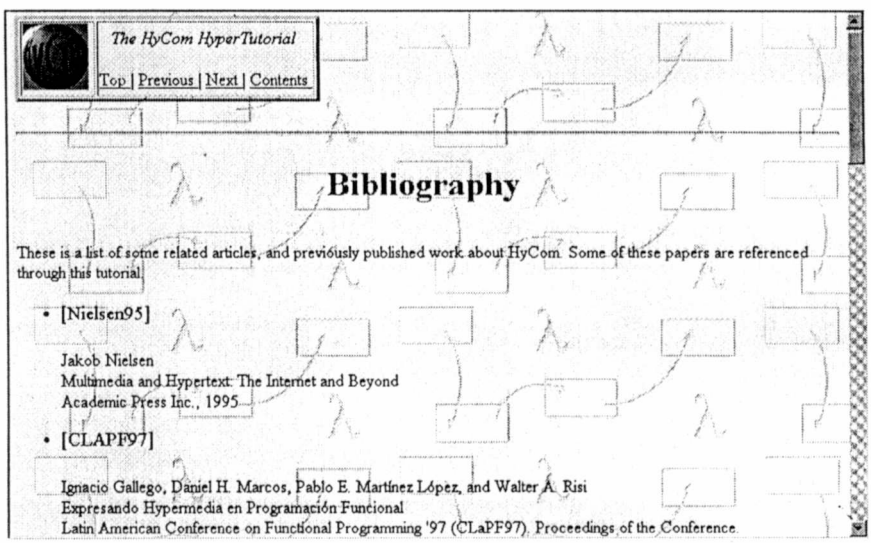

Figura C.10: Bibliografía del *Tutorial* del HyCom 1.0

Figura C.11: *Home Page* del Grupo BioCom

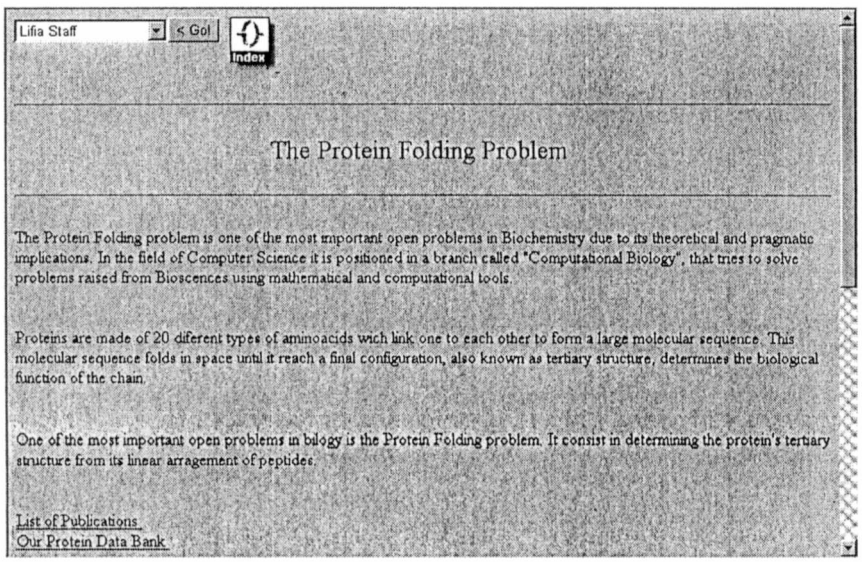

Figura C.12: Página sobre el Problema del Plegado de Proteínas

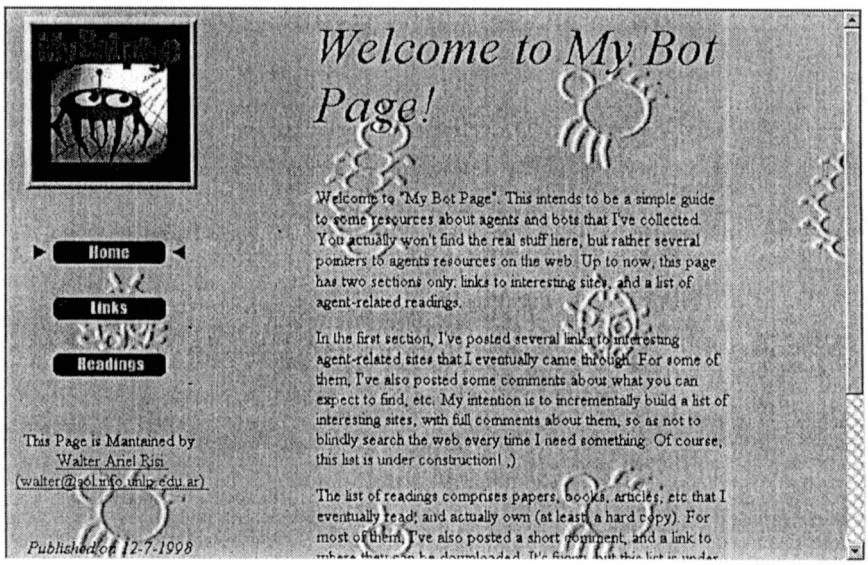

Figura C.13: *Home Page* sobre Agentes

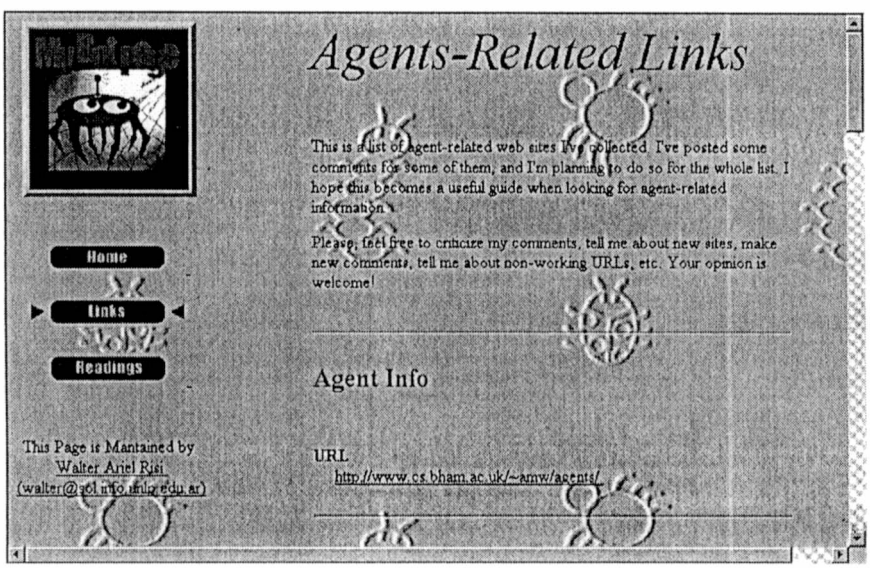

Figura C.14: Página con *Links* Relacionados a la Tecnología de Agentes

Bibliografía

Bibliografía

[Ash96] Mohammad Ashrafuzzaman. Conceptual Modeling of a Hypermedia Design Environment in Metaview, January 1996. Course Project CMPT856: Topics in Software and Knowledge Engineering.

[ASP99] Microsoft Active Server Pages. `http://www.webnations.com/scotts/ASPpages/mainasp.asp`, 1999.

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin*, pages 218–27, August 1984.

[BBI98] V. Balasubramanian, Michael Bieber, and Tomás Isakowitz. Systematic Hypermedia Design. In *Proceedings of Information Systems Journal (ISJ98)*, 1998.

[Ben86] Jon Bentley. Little Languages. *Communications of the ACM*, 29(8):711–721, January 1986.

[BI95] M. Bleber and T. Isakowitz, editors. *Communications of the ACM*, volume 38 (8). ACM Press, August 1995.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Press, 1998. Second Edition.

[BN96] Martin Bichler and Stefan Nussler. Modular Design of Complex Web-Applications with W3DT. In *Proceedings of the 5th Workshop of Enabling Technologies: Infrastructure of Collaborative Enterprises (WET ICE '96)*, Stanford, 1996. IEEE Computer Press.

[Boo91] G. Booch. *Object Oriented Design with Applications.* Benjamin Cummings, 1991.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[CH98] Magnus Carlsson and Thomas Hallgren. *Fudgets: Purely Functional Processes with applications to Graphical User Interfaces.* Department

of Computing Science, Chalmers University of Technology, 1998. Phd Thesis.

[CHJ93] W. E. Carlson, P. Hudak, and M. P. Jones. An Experiment Using Haskell to Prototype "Geometrical Regions Servers" for Navy Command and Control. Technical report, Department of Computer Science, Yale University, November 1993.

[CL99] D. D. Cowan and C. J. P. Lucena. Abstract Data Views, An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering*, 21(3), March 1999.

[Cle99] The Clean Home Page.
`http://www.cs.kun.nl/~clean/`, 1999.

[Col99] Allaire Corporation Home Page.
`http://www.coldfusion.com/`, 1999.

[COM99] Microsoft COM Specification.
`http://channels.microsoft.com/com/resources/comdocs.asp`, 1999.

[COR99] OMG: CORBA Specification.
`http://www.ti5.tu-harburg.de/Manual/OMG/CORBA/index.htm`, 1999.

[CSS99] Cascading Style Sheets.
`http://www.w3.org/Style/css/`, 1999.

[Dav92] Antony J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.

[DCO99] Microsoft DCOM Specification.
`http://www.microsoft.com/com/tech/DCOM.asp`, 1999.

[DI95] A. Díaz and T. Isakowitz. RMCase: A Computer-Aided Support for Hypermedia Design and Development. In *International Workshop on Hypermedia Design*. Springer-Verlag, 1995.

[EH97] Conal Elliot and Paul Hudak. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP97)*, 1997.

[EN90] Ramez Elmasri and Shamkant Navate. *Fundamental of Database Systems.* The Benjamin/Cummings Publishing Company, 1990. Second Edition.

[FNN96] S. Fraïssé, Jocelyne Nanard, and Marc Nanard. Generating Hypermedia from Specifications by Sketching Multimedia Templates. In *Proceedings of the ACM Multimedia '96*, 1996.

[FPJ96] S. Finne and S. Peyton Jones. Composing the User Interface with Haggis. In Launchbury et al. [LMS96], pages 1–37.

[Fro99] FrontPage Home Page, Microsoft Corporation. `http://www.microsoft.com/FrontPage/`, 1999.

[GBG98] M. Gaedke, M. Beigl, and H. W. Gellersen. Mobile Information Access: Catering for Heterogeneous Browser Platforms. In *Proceedings of the International Workshop on Mobile Data Access in Conjunction with 17th International Conference on Conceptual Modelling (ER98)*, Singapore, 1998.

[GH95] Andrew D. Gordon and Kevin Hammond. Monadic I/O in Haskell 1.3. In Paul Hudak, editor, *Proceedings of the Haskell Workshop*, pages 50–69, La Jolla, California, June 1995.

[GHC99] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GR83] Adele J. Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis.* Englewood Cliffs, Prentice-Hall, 1979.

[GWG97] H. W. Gellersen, R. Wicke, and M. Gaedke. WebComposition: An Object-Oriented Support System for the Web Engineering Lifecycle. In *Computer Networks and ISDN Systems 29, Special Issue of the 6th World-Wide Web Conference*, Santa Clara, CA, USA, 1997.

[Has99] The Haskell Home Page. `http://www.haskell.org/`, 1999.

[HBI97] A. M. Hester, R. C. Borges, and R. Ierusalimsky. CGILua: A Multi Paradigmatic Tool for Creating WWW Pages. In *Proceedings of the XI Brazilian Software Engineering Symposium (SBES '97)*, Fortaleza, Brasil, 1997.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In Jeuring and Meijer [JM95], pages 137–182.

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical report, University of Nottingham, 1996.

[HS88] Paul Hudak and Raman S. Sundaresh. On the Expressiveness of Purely Functional I/O Systems. Technical report YALEU/DCS/RR665, Yale University, Department of Computer Science, December 1988.

[HTM99] Hypertext Markup Language Home Page.
`http://www.w3.org/MarkUp/`, 1999.

[Hud96a] Paul Hudak. Building Domain-Specific Embedded Languages. Technical report, Department of Computer Science, Yale University, June 1996.

[Hud96b] Paul Hudak. Haskore Music Tutorial. In Launchbury et al. [LMS96], pages 38–67.

[Hug89] John Hughes. Why Functional Programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Hug99] John Hughes. Generalizing Monads to Arrows.
To appear in Science of Computer Programming. Available at `http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps`, 1999.

[IKK97a] T. Isakowitz, A. Kamis, and M. Koufaris. Extending The Capabilities of RMM: Russian Dolls and Hypertext. In *Proceedings of the 30th Annual Hawaii International Conference on System Sciences*, 1997.

[IKK97b] T. Isakowitz, A. Kamis, and M. Koufaris. Reconciling Top-Down and Bottom-Up Design Approaches in RMM. In *Proceedings of the Workshop on Information Technologies and Systems (WITS-97)*, Atlanta, GA, December 1997.

[IKK98] T. Isakowitz, A. Kamis, and M. Koufaris. The Extended RMM Methodology for Web Publishing. 1998.

[ISB95] T. Isakowitz, E. A. Stohr, and P. Balasubramaninan. RMM: A Methodology for Structured Hypermedia Design. [BI95], pages 34–44.

[Jac94] Ivar Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley Publishing Company, 1994.

[Jav99] JavaBeans Specification.
`http://java.sun.com/beans/spec.html`, 1999.

[JM95] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming, LNCS 925.* Springer-Verlag, May 1995.

[Jon95] Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Jeuring and Meijer [JM95], pages 97–136.

[Jon99] Simon L. Peyton Jones. Explicit Quantification in Haskell. `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`, 1999.

[KK97] Peter Kent and John Kent. *Official Netscape Javascript 1.2 Book: The Nonprogrammer's Guide to Creating Interactive Web Pages.* Netscape Press, 1997. Second Edition.

[LM99] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. Submitted to 2nd USENIX Conference on Domain-Specific Languages, 1999.

[LMS96] John Launchbury, Erik Meijer, and Tim Sheard, editors. *Advanced Functional Programming, LNCS 1129.* Springer-Verlag, August 1996.

[LR96] Fernando D. Lyardet and Gustavo Rossi. Bridging the Gap Between Design and Implementation. A Research Prototype. In *Proceedings of Hypertext'96*, Washington, USA, 1996. Poster.

[Mar82] J. Martin. *Application Development without Programmers.* Englewood Cliffs, Prentice-Hall, 1982.

[MLH98] Erik Meijer, Daan Leijen, and James Hook. Client-side Web Scripting with HaskellScript. In *Proceedings of Practical Aspects of Declarative Languages (PADL)*, 1998.

[MMLR97] Daniel H. Marcos, Pablo E. Martínez López, and Walter A. Risi. Expresando Hypermedia en Programación Funcional. In *Proceedings of the Second Latin American Conference on Functional Programming (CLaPF97)*, La Plata, Buenos Aires, Argentina, 1997.

[MMLR98] Daniel H. Marcos, Pablo E. Martínez López, and Walter A. Risi. A Functional Programming Approach to Hypermedia Authoring (Poster). In *Proceedings of ACM International Conference on Functional Programming (ICFP98)*, Baltimore, Maryland, USA, 1998.

[Net99] NetObjects Home Page. `http://www.NetObjects.com`, 1999.

[Nie95] Jakob Nielsen. *Multimedia and Hypertext: The Internet and Beyond.* Academic Press Inc., 1995.

[NN95] Jocelyne Nanard and Marc Nanard. Hypertext Design Enviroments and the Hypertext Design Process. [BI95], pages 49–56.

[ODB99] Microsoft ODBC. `http://www.microsoft.com/data/odbc/`, 1999.

[PJH99] Simon Peyton Jones and John Hughes. Report on the Programming Language Haskell '98. Technical report, Yale University, February 1999. Available online: `http://www.haskell.org/report`.

[PJJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type Classes: An Exploration of the Design Space. To appear on the Haskell Workshop 1997, 1997.

[PV95] Peter Pauen and Josef Voss. The HyDev Approach to Model-Based Development of Hypermedia Applications. In *ACM Workshop on Effective Abstractions in Multimedia*, November 1995.

[R+91] J. Rumbaugh et al. *Object Oriented Modelling and Design*. Prentice-Hall, 1991.

[RMML99] Walter A. Risi, Daniel H. Marcos, and Pablo E. Martínez López. Effective Mapping of Hypermedia High-Level Design Primitives to Implementation Environments. In *Proceedings of V Congreso Argentino de Ciencias de la Computación (CACiC99)*, Tandil, Buenos Aires, Argentina, 1999.

[Ros95] Gustavo H. Rossi. *An Object Oriented Hypermedia Design Method (OOHDM)*. PhD thesis, Departamento de Informática, PUC-Rio, Rio de Janeiro, 1995.

[RSG97] G. Rossi, D. Schwabe, and A. Garrido. Design Reuse in Hypermedia Design Application Development. In *Proceedings of the ACM International Conference on Hypertext (HT97), Southampton*. ACM Press, April 1997.

[SB94] Daniel Schwabe and Simone D. J. Barbosa. Navigation Modelling in Hypermedia Applications. Technical report, Departamento de Informática, PUC-Rio, Rio de Janeiro, 1994.

[SdAP99] Daniel Schwabe and Rita de Almeida Pontes. A Method-Based Web Application Development Environment. In *Web Engineering Workshop (WWW8)*, 1999.

[SR95] D. Schwabe and G. Rossi. The Object-Oriented Hypermedia Design Model. [BI95], pages 45–46.

[Szy98] Clemens Szypersky. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley Publishing Company, January 1998.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

[TM98] Yin Leng Theng and Gil Marsden. Authoring Tools: Towards Continuous Usability Testing of Web Documents. In *Proceedings of the 1st International Workshop on Hypermedia Development, Held in Conjunction with Hypertext '98*, 1998.

[U+87] D. Ungar et al. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87.*, 1987.

[vDM96] Joost van Dijk and Erik Meijer. CGI Programming in Gofer. In *Proceedings of the First Latin-American Workshop on Functional Programming*, September 1996.

[Wad95] Philip Wadler. Monads for functional programming. In Jeuring and Meijer [JM95].

[Web99] WebObjects Home Page. `http://www.webobjects.com/`, 1999.

[XML99] Extensible Markup Languaje (XML). `http://www.w3.org/XML/`, 1999.

[You91] E. Yourdon. *Modern Structured Analysis.* Englewood Cliffs, Prentice-Hall, 1991.