Expresando Hypermedia en Programación Funcional

Ignacio Gallego Sagastume   Daniel H. Marcos   Pablo E. Martínez López
Walter A. Risi

LIFIA, Departamento de Informática, Universidad Nacional de La Plata.
C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.
E-mail: {nick,daniel,fidel,walter}@lifia.info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar/

Resumen

En este artículo se presentan un conjunto de herramientas construídas sobre el lenguaje funcional *lazy* Haskell para describir documentos *hypermediales*. El trabajo consiste en el diseño e implementación de un conjunto de funciones y tipos que permitan describir los elementos básicos de un documento *hypermedial* (nodos, *links*, componentes multimediales y la interacción entre ellos). La construcción de los documentos de *hypermedia* se realiza utilizando una metodología composicional, en la que componentes de *hypermedia* complejos se construyen combinando repetidamente otros más simples. Esta metodología puede utilizarse también para crear componentes genéricos y reusables. Otro aspecto interesante es que los documentos descritos con estas herramientas no dependen de construcciones específicas de una plataforma de *hypermedia* en particular. Consecuentemente, los documentos construídos de esta forma pueden ser trasladados luego a diferentes plataformas de *hypermedia*, como HTML o Ayuda de Windows, sin necesidad de efectuar modificaciones.

Expresando Hypermedia en Programación Funcional

1 Introducción

No hay duda de que los sistemas *hypermediales* [Nie95] han sufrido gran expansión en los últimos años. Existen múltiples razones que justifican este desarrollo, aunque probablemente la más importante es que estos sistemas trabajan en forma interactiva, brindando un formato muy amigable de presentar información al usuario. Ésto, unido con el abaratamiento de los medios necesarios para soportar facilidades multimediales e *hypermediales* (lectores de CD-ROMs, terminales gráficas, etc.), ha conducido a que cada vez sean más los que se inclinan a utilizar medios *hypermediales* en vez de los tradicionales (medios impresos por lo general). Por otro lado, la expansión de la WWW ha generado un campo muy amplio para el desarrollo de aplicaciones *hypermediales* distribuídas sobre redes en gran escala.

La programación funcional [BW88] ofrece un gran número de ventajas a los programadores, por ejemplo un alto nivel de abstracción y una gran capacidad de expresión. Estas características surgen de los poderosos mecanismos de abstracción presentes en los lenguajes funcionales (funciones de alto orden, polimorfismo, etc.). Estas ventajas se reflejan en un tiempo de desarrollo de programas menor que el requerido habitualmente con lenguajes imperativos, resultando ser además el código más fácil de entender, mantener y reusar.

En este artículo se presentan un conjunto de herramientas construídas sobre el lenguaje funcional *lazy* Haskell [PH+96] para describir documentos *hypermediales*. El trabajo consiste en el diseño e implementación de un conjunto de funciones y tipos que permiten describir los elementos básicos de un documento *hypermedial* (nodos, *links*, componentes multimediales y la interacción entre ellos).

El trabajo resulta novedoso pues utiliza el paradigma funcional para describir *hypermedias*; estas tareas han estado muy distanciadas en el ambiente informático de Latinoamérica, y es la intención principal de este artículo acercar ambas disciplinas. Además, otros aspectos atractivos del trabajo son la separación que se obtiene entre la descripción semántica de un documento *hypermedial* y su representación visual en una plataforma determinada, y la facilidad de definir componentes de *hypermedia* genéricos reusables. Es importante destacar la utilización de una metodología composicional, en la que componentes de *hypermedia* complejos pueden construírse combinando componentes más simples. Finalmente, el trabajo es interesante como aplicación de técnicas de diseño de programación funcional; ejemplos de las mismas son el uso de combinadores y transformadores.

El artículo comienza con una breve introducción a los conceptos básicos de *hypermedia* que se utilizan a lo largo del trabajo. En la Secc. 3 se presenta el diseño general de la aplicación. La Secc. 4 describe como se realiza la traducción de un documento descrito con las herramientas de la sección anterior a una plataforma de *hypermedia* específica. En la Secc. 5 se presentan algunos ejemplos, y en la Secc. 6 se explican los detalles de implementación. Finalmente, en la

Secc. 7 se mencionan algunos trabajos relacionados.

2 Conceptos básicos sobre hypermedia

Puede definirse *hypermedia* como la conjunción de *multimedia* e *hypertexto*. Se dice que una aplicación es *multimedial* cuando reúne dentro de ella varios medios, como texto, imágenes, video, audio, etc. Por otra parte, la manera más simple de definir *hypertexto* es haciendo una comparación con los medios de información tradicionales, como libros y revistas. En un formato tradicional, la información se presenta en forma secuencial, es decir, existe una secuencia que define el orden en que la información debe ser considerada. En un *hypertexto*, por otro lado, el texto se agrupa en unidades de información relacionada; estas unidades son llamadas *nodos*, y a su vez éstos pueden referenciarse unos a otros. Es importante notar que el usuario no accede a la información en forma secuencial, sino que lo hace siguiendo las referencias a otros nodos.

Las referencias que conectan dos nodos (que pueden ser en ambos sentidos a la vez) se llaman *links* o *hyperlinks*. El nodo del cual parte el *link* se denomina *origen del link*. El nodo o parte del nodo al cual se llega a través de la activación de un *link*, se denomina *destino del link*.

Los *hyperlinks* están asociados con partes específicas de los nodos que conectan, como por ejemplo una palabra o una imagen. Estas partes de los nodos asociadas con *links* se denominan *anchors* (anclas), y determinan, o bien el origen de un *link* (una parte del nodo que permite activar el link), o bien el destino (una parte particular del nodo destino a la que se llega a través de la activación del *link*). Los *anchors* del primer grupo se denominan *anchors de origen*, y los del segundo grupo *anchors de destino*.

Muchas veces se hace distinción entre distintos tipos de *links*. Por ejemplo, se suele diferenciar entre *links* que hacen referencia a cierta parte dentro del mismo nodo, los que lo hacen a otro nodo dentro del mismo documento, y los que referencian a un nodo perteneciente a otro documento *hypermedial*.

Ejemplos de sistemas *hypermediales* son la *World Wide Web*, *Hyper-G* y la Ayuda de Windows.

3 Descripción del Trabajo

El trabajo consiste en un conjunto de abstracciones que representan los elementos de *hypermedia* existentes en la mayoría de los sistemas actuales. A continuación se presenta la formalización de estos elementos dentro de este trabajo, junto con las abstracciones que los representan.

3.1 Nodos

Los nodos son las unidades básicas de información que constituyen los sistemas *hypermediales*. Un nodo está constituído internamente por bloques de información expresados en un medio

particular (imagen, texto, etc.). Además de dichos bloques, los nodos poseen estructuras cuya funcionalidad es la de permitir conectarse a otros nodos.

En este modelo la abstracción que representa este concepto es el tipo `Node`. La forma de construir un nodo es a través de la función de construcción `node`.

```
node :: NodeId -> HG -> Node
```

Un nodo está compuesto por un identificador de nodo, `NodeId`, y un componente, `HG` (*Hypergadget*). El `NodeId` es un elemento que permite identificar unívocamente a un nodo (ver Secc. 3.5). Los *Hypergadgets* son los componentes mediante los cuales se constituye internamente un nodo (ver Secc. 3.3).

Existen funciones que denominamos *transformadores de nodos* que al aplicarse a un nodo, le agregan alguna característica que lo afecta en un aspecto particular. Algunas transformaciones típicas son, por ejemplo:

```
defaultTextColor, backgroundColor :: Node -> Color -> Node
```

La transformación `defaultTextColor` determina que el nodo transformado tendrá por color de texto por defecto el que se pasa como parámetro. En forma similar, la transformación `backgroundColor` determina el color de fondo del nodo al cual se aplica.

3.2 Hyperlinks

Los *hyperlinks* son los elementos que permiten relacionar varios nodos entre sí. Los *links* junto con los nodos son los elementos que estructuran los sistemas *hypermediales*; en particular, a partir de la existencia de los *links* surge la actividad denominada "navegación", la cual hace referencia al modo en que el usuario accede a la información dentro del sistema *hypermedial*.

En este trabajo se distingue entre los *links* que conectan nodos que pertenecen a un mismo documento *hypermedial* y los que conectan nodos pertenecientes a distintos documentos; esta distinción pretende marcar diferencias temáticas o conceptuales antes que de ubicación física. Se denomina al primer grupo *Near Hyperlinks* y al segundo grupo *Far Hyperlinks*. Los tipos que los representan son `NearHL` y `FarHL` respectivamente. Existe además el tipo general `HL` (*hyperlink*), cuyos elementos representan *hyperlinks*, independientemente de que sean *near* o *far*.

La clase `Hyperlink` agrupa a los tipos que representan *hyperlinks*. Los tipos que pertenecen a esta clase son `HL`, `NearHL` y `FarHL` (para más detalles ver la Secc. 6.1).

El destino de un *link* puede ser o bien un nodo en su totalidad, o bien una parte específica de un nodo (un párrafo de texto determinado, una imagen, etc.). Además, el nodo destino y el nodo origen pueden ser el mismo nodo, con lo cual un nodo podría tener un link a otra parte de sí mismo. Por lo tanto, surge la necesidad de tener una manera de especificar el destino de un *link*. En este trabajo se considerará que un *link* contiene un destino, representado por

la abstracción HLDestiny (*Hyperlink Destiny*), la cual contendrá la información requerida para identificar al nodo destino, o a la parte del mismo a la que se quiere llegar.

Se tienen funciones de construcción para *hyperlinks*:

```
nearHL :: HyperlinkDestiny a => a -> NearHL
farHL  :: HyperlinkDestiny a => a -> FarHL
```

La clase HyperlinkDestiny agrupa a aquellos tipos cuyos elementos pueden ser utilizados para denotar el destino de un *hyperlink*. Los elementos de tipo Node o NodeId pertenecen a esta clase, ya que pueden utilizarse para indicar que ese nodo es el destino del *hyperlink*. El mecanismo con el cual se construyen los *Hyperlink Destinys* se verá con más profundidad en la Secc. 6.1.1.

3.3 Hypergadgets

Un nodo está compuesto internamente por componentes que definen su estructura. Ejemplos de estos componentes son bloques de texto, imágenes, listas, *anchors*, etc. En este trabajo se denomina *Hypergadget* a un componente como los mencionados. Se hace diferencia entre los *Hypergadgets* puramente visuales, llamados *Dumb Hypergadgets*, y aquellos que además son *anchors* de origen o destino de un *link*, denominados *Anchor Hypergadgets*.

El tipo de los *Dumb Hypergadgets* es DumbHG, y el tipo de los *Anchor Hypergadgets* es AnchorHG. Existe además el tipo general HG, que representa un *Hypergadget*, independientemente de que sea un *Dumb Hypergadget* o *Anchor Hypergadget*.

Existen *Hypergadgets* atómicos que representan elementos simples, como un bloque de texto, una imagen, una lista, etc. Componentes como los mencionados pueden construirse a partir de las funciones correspondientes.

```
txtDumbHG      :: String -> TxtDumbHG
imgDumbHG      :: ImgSource -> ImgDumbHG
itemListDumbHG :: Hypergadget h => [h] -> ItemListDumbHG
```

Otros *hypergadgets* atómicos son los *anchors*. Existen *anchors* de origen (aquellos de los que parte un *link*) y *anchors* de destino (aquellos a los que se llega a partir de un *link*). El tipo de los primeros es OrgAnchorHG y el tipo de los segundos es DstAnchorHG. El tipo general AnchorHG representa un *anchor*, independientemente de que sea de origen o destino. Los *anchors* tienen además una parte visual. Las funciones de construcción correspondientes son orgAnchorHG y dstAnchorHG.

```
orgAnchorHG :: (DumbHypergadget d, Hyperlink h) => d -> h -> OrgAnchorHG
dstAnchorHG :: Hypergadget h => DstAnchorId -> h -> DstAnchorHG
```

Un *anchor* de origen se construye a partir de un *Dumb Hypergadget* (elemento de un tipo perteneciente a la clase DumbHypergadget) y de un *link* (elemento de un tipo de la clase Hyperlink).

El *Dumb Hypergadget* constituye la parte visual del *anchor*, y el *link* es aquél del cual el *anchor* es origen.

Un *anchor* de destino se construye a partir de un *Hypergadget* (elemento de un tipo perteneciente a la clase `Hypergadget`) y de un identificador de *anchor* destino, `DstAnchorId`, que identifica unívocamente al *anchor* de destino (ver Secc. 3.5).

Los elementos de los tipos `DstAnchorHG` y `DstAnchorId` pueden ser utilizados para indicar el destino de un *link*, por lo tanto, pertenecen a la clase `HyperlinkDestiny`.

La clase de tipo `Hypergadget` agrupa a los tipos que denotan *Hypergadgets*. Además, las clases `DumbHyperGadget` y `AnchorHypergadget`, subclases de la anterior, agrupan a los tipos que representan *Dumb Hypergadgets* y *Anchor Hypergadgets*, respectivamente (ver Secc. 6.2 para detalles de implementación).

Los tipos `DumbHG`, `TxtDumbHG`, `ImgDumbHG` y `ItemListHG` son instancias tanto de la clase `Hypergadget` como `DumbHypergadget`. De la misma manera, los tipos `AnchorHG`, `OrgAnchorHG` y `DstAnchorHG` son instancias tanto de la clase `Hypergadget` como `AnchorHypergadget`. Otra instancia de la clase `Hypergadget` es el tipo `HG`.

Se presenta a continuación un pequeño ejemplo en donde se utilizan las funciones de construcción anteriormente vistas. Lo que se define es un *anchor* de origen, `toAnchor`, y un *anchor* destino, `anchorDst`, el cual representa una lista de elementos. La expresión `thisNodeId` simplemente se corresponde al identificador del nodo al que se está estructurando.

```
-- Definicion de Dumb Hypergadgets
item1     = txtDumbHG "Item1"
item2     = txtDumbHG "Item2"
lista     = itemListDumbHG [item1, item2]

-- Definicion de hyperlinks y Anchor Hypergadgets
anchorDst = dstAnchorHG (dstAnchorId "anchorDst" thisNodeId) lista
nearLink  = nearHL anchorDst
toAnchor  = orgAnchorHG (txtDumbHG "Observar la lista ...") nearLink
```

3.3.1 Combinación y transformación de Hypergadgets

Un *combinador de Hypergadgets* es una función que toma dos o más *Hypergadgets*, devolviendo uno más complejo que resulta de la combinación de los mismos. Dicha combinación se hace siguiendo reglas de *layout*, y algunos combinadores son, por ejemplo:

```
aboveOf, belowOf,
leftOf, rightOf,
next :: (Hypergadget a, Hypergadget b) => a -> b -> HG
```

La combinación `aboveOf`, por ejemplo, toma dos *Hypergadgets*, y retorna uno nuevo que resulta de ubicar el primero arriba del segundo. Los demás combinadores se comportan de manera análoga.

Los operadores `(/=\)`, `(\=/)`, `(<=<)`, `(>=>)` y `(+++)`, son equivalentes a `aboveOf`, `belowOf`, `leftOf`, `rightOf` y `next`, respectivamente. La elección de órdenes de precedencia adecuados reduce el uso de los paréntesis necesarios al escribir expresiones usando estos combinadores.

```
infixr 6 <=<, >=>
infixr 5 /=\, \=/
infixr 4 +++
```

Algunos *Dumb Hypergadgets* interesantes que se utilizan junto con los combinadores de *layout* mencionados son `hSpace` y `vSpace`.

```
hSpace, vSpace :: Int -> DumbHG
```

Las funciones `hSpace` y `vSpace` sirven para dejar espacios (horizontales y verticales, respectivamente) de una cierta cantidad de pixels. Estos componentes se utilizan exclusivamente por razones de *layout*.

Otro *Hypergadget* importante es el *Hypergadget nulo*. Este es el elemento neutro para cualquiera de las combinaciones vistas anteriormente. La función `nullHG` permite crear un elemento de este tipo.

```
nullHG :: DumbHG
```

Un *transformador de Hypergadgets* es una función que al aplicarse a un *Hypergadget* le agrega alguna característica que lo afecta en algún aspecto particular. Algunas de las transformaciones típicas son:

```
txtColor                       :: Color -> TxtDumbHG -> TxtDumbHG
justifyC, justifyR, justifyL :: HG -> HG
```

La transformación `txtColor` toma un *Hypergadget* de texto, `TxtDumbHG`, y le asigna un color particular. Las transformaciones `justifyC`, `justifyR` y `justifyL` hacen que el *Hypergadget* transformado esté alineado al centro, derecha o izquierda, respectivamente.

Un transformador interesante es `txtWithAnchors`, el cual provee una forma práctica de insertar *anchors* de origen en un componente de texto. Esta función dos argumentos: una lista de asociaciones entre `Strings` e `hyperlinks` y un *Hypergadget* de texto. El resultado de aplicar esta función es un componente de tipo `HG`, en donde los componentes de texto que figuran en la lista son reemplazados por *anchors* de origen con los correspondientes *links*.

```
txtWithAnchors :: Hyperlink h => [(String, h)] -> TxtDumbHG -> HG
```

La utilización de la función anterior evita el tener que definir los *anchors* en forma separada, y luego combinarlos mediante el operador `(+++)`.

3.4 Documentos hypermediales

Un documento *hypermedial* es una colección de nodos relacionados entre sí por *links*. En este trabajo la abstracción que representa este concepto es el tipo `HD` (*hypermedia document*), y la forma de construir un documento es a través de la función de construcción `hd`.

```
hd :: HDId -> [Node] -> HD
```

Un documento *hypermedial* está compuesto por un identificador de documento, `HDId`, y un conjunto de nodos. Un `HDId` es un elemento que permite identificar unívocamente a un documento *hypermedial* (ver Secc. 3.5).

3.5 Identificadores

Algunos componentes requieren ser identificados unívocamente dentro del la *hypermedia* que se está construyendo. Para esto se utilizan estructuras denominadas *identificadores*.

Un identificador de tipo `NodeId` permite identificar en forma unívoca a un nodo. Un elemento de este tipo se construye mediante la función `nodeId`.

```
nodeId :: String -> HDId -> NodeId
```

El `String` con el cual se construye el `NodeId` es un nombre que debe ser único para cada nodo en un mismo documento, pudiéndose repetir en nodos de distintos documentos *hypermediales*.

El identificador de *anchor* de destino está representado por el tipo `DstAnchorId` y los elementos de este tipo se construyen mediante la función `dstAnchorId`.

```
dstAnchorId :: String -> NodeId -> DstAnchorId
```

El `String` con el cual se construye el `DstAnchorId` debe ser único dentro del nodo en donde se ubica el `DstAnchor` en cuestión.

El tipo `HDId` representa al identificador de *documento hypermedial* y los elementos de este tipo se construyen mediante la función `hdId`.

```
hdId :: String -> HDId
```

El `String` con el cual se construye el `HDId` debe ser único de ese documento.

4 Portabilidad a diferentes plataformas

Las construcciones provistas en la Secc. 3 permiten expresar la estructura y la apariencia de un documento *hypermedial* sin hacer uso de características específicas de una plataforma de *hypermedia* particular. De esta forma, un documento expresado con las construcciones ya presentadas

puede ser trasladado sin modificaciones a cualquier plataforma de *hypermedia* (HTML, Ayuda de Windows, etc.), utilizando *funciones de traducción*.

Las funciones de traducción a una plataforma específica se proveen en módulos que el usuario importa dependiendo de la plataforma que elija. El módulo de una plataforma no sólo provee las funciones de traducción, sino también estructuras y funciones que permiten incorporar características propias de la misma; estas características no están incluídas en el modelo base por ser muy específicas de una plataforma particular.

Supóngase que se decide traducir el documento a la plataforma HTML. Lo primero que debe hacerse es importar el módulo de la plataforma HTML.

```
import HTML
```

El módulo `HTML` provee la función de traducción `renderHD`. La misma convierte un documento *hypermedial* construído con las estructuras vistas, a un conjunto de archivos HTML.

```
type PageNames = [(NodeId, String)]

renderHD :: HD -> PageNames -> IO ()
```

La función `renderHD` toma un documento *hypermedial* y una lista de pares que asocia a cada nodo un nombre de archivo. El resultado de evaluar la función es un conjunto de archivos HTML. Nótese que la función anterior utiliza entrada/salida monádica [Wad95, GH95].

Otra función de traducción provista por el módulo `HTML` es `renderHDs`, que permite la traducción de varios documentos.

```
renderHDs :: [(HD, PageNames)] -> IO ()
```

La función `renderHDs` es necesaria cuando se definen documentos con *hyperlinks* entre sí.

El módulo `HTML` provee además funciones que permiten incorporar construcciones específicas de la plataforma HTML. Un ejemplo es el transformador de nodo `backgroundImg`.

```
backgroundImg :: ImgSource -> Node -> Node
```

La función `backgroundImg` toma un elemento `ImgSource` (especifica donde está el archivo de imagen) y un elemento `Node`, y devuelve un nodo que tiene como fondo a la imagen especificada. El tipo `ImgSource` es una construcción que permite especificar el archivo fuente de una imagen. Cada plataforma provee una definición para este tipo, por ejemplo, en la plataforma HTML un elemento del tipo `ImgSource` corresponde a un URL.

```
type ImgSource = URL
```

El tipo `URL` es específico de la plataforma HTML. Un elemento de este tipo puede construírse mediante la función `url`.

```
url :: String -> URL
```

El elemento de tipo `String` que recibe la función corresponde a un URL.

5 Ejemplos

A continuación, se presentan tres ejemplos del uso de las estructuras descritas en este trabajo. El primero muestra la forma de construir un documento *hypermedial* sencillo con las estructuras vistas; el segundo ejemplo ilustra como las herramientas provistas son útiles para definir nuevos componentes reusables a partir de los existentes; en el tercer y último ejemplo, se ejemplifica el uso de *hyperlinks* para conectar ambos documentos.

Nótese que en estos ejemplos se utiliza HTML como la plataforma destino, por lo cual se utilizan algunas construcciones propias del módulo específico de la plataforma HTML.

5.1 Un Documento Simple

Supóngase que se desea escribir una página de WWW que brinde información básica sobre la *2da Conferencia Latinoamericana de Programación Funcional*. Con esta aplicación, se modelará como un documento *hypermedial* con un único nodo.

Se comienza definiendo un componente de texto que será el título de la página. Se desea que el título se destaque del resto del texto; para esto, se utiliza la transformación `txtLevel` con el parámetro `H1`, lo cual indica que se trata de un título de importancia. Aparte de esta transformación, se utiliza `txtColor`, para que el texto resultante se muestre en color rojo, y la transformación `justifyL` para que el componente esté alineado a la izquierda.

```
title str = justifyL $ txtLevel H1 $ txtColor Red $ txtDumbHG str
```

El operador `$` es simplemente aplicación de funciones; se lo utiliza en el ejemplo para evitar el uso excesivo de paréntesis.

El siguiente paso consiste en definir un subtítulo.

```
subTitle str = justifyL $ txtLevel H2 $ txtColor Red $ txtDumbHG str
```

Obsérvese que en este caso el transformador `txtLevel` lleva como parámetro adicional `H2`, lo cual indica que el componente de texto en cuestión es un título de menor importancia que el anterior.

La siguiente tarea consiste en escribir un párrafo de introducción. Además, se desea que el dicho párrafo contenga un *hyperlink* a la página de la primera edición de la conferencia, para lo cual habrá que definir dicho *link* e insertar un *anchor* en que lo active.

```
clapfIntro = txtWithAnchors [('primera edicion', toFirstEd)] $ txtDumbHG
                     "Estamos orgullosos de anunciar la organizacion de la 2da
```

```
                    Conferencia Latinoamericana de Programacion Funcional. El
                    exito de la primera edicion nos alienta a continuar
                    trabajando para lograr las metas que nos guiaron: "
```

El *link* `toFirstEd` se define como sigue:

```
toFirstEd = farHL $ url
               "http://www-lifia.info.unlp.edu.ar/~lambda/first/spanish"
```

El *hyperlink* anterior es un `FarHL` ya que sale del documento corriente. El destino del *link* se especifica mediante un *URL* (que puede incluirse usando la función `url`, específica de la plataforma HTML) ya que el documento en cuestión no está definido dentro de esta aplicación, sino que existe como documento HTML independiente.

Para completar el documento, se desea crear una lista de ítems con las razones principales que motivan la realización de la conferencia. Se utiliza para esto el componente `itemListHG`.

```
aimList = itemListDumbHG [item1, item2, item3]
item1   = txtDumbHG "Promover el uso de la programacion funcional en ambientes
                     academicos y no academicos."
item2   = txtDumbHG "Promover el acercamiento entre los investigadores y el
                     intercambio de conocimientos y experiencias."
item3   = txtDumbHG "Dar a conocer las experiencias de aplicacion practica y
                     las posibilidades que ofrece la programacion funcional."
```

Definidos todos los componentes, sólo resta combinarlos todos en el cuerpo del nodo.

```
clapfBody =
   title "2da Conferencia Latinoamericana de Programacion Funcional" /=\
   subtitle "3 y 4 de Octubre de 1997, La Plata, Argentina"          /=\
   divisionLine                                                      /=\
   clapfIntro                                                        /=\
   aimList                                                           /=\
   divisionLine
```

El componente `divisionLine` es una línea divisoria, que se incluye para mejorar la presentación del documento.

El paso restante es definir un nodo con el cuerpo, `clapfBody`, y un documento *hypermedial* que contenga a ese nodo.

```
clapfNodeId = nodeId "CLapF Home Page" clapfDocId
clapfNode   = backgroundImg (url "fondo1.gif") $ defaultTxtColor Black $
                 node clapfNodeId clapfBody
```

```
clapfDocId  = hdId "Home Page De CLaPF"
clapfDoc    = hd clapfDocId [clapfNode]
```

Nótese que al nodo definido se le aplican dos transformaciones. La transformación `backgroundImg` (específica de la plataforma HTML) determina que se utilizará el archivo `fondo1.gif` como gráfico de fondo para la página. La transformación `defaultTxtColor` determina que el color de texto por defecto será negro.

Se tiene entonces un documento simple que cuenta con un único nodo. Supóngase que se desea transladar este documento a plataforma HTML; se utiliza para esto la función `renderHD` del módulo `HTML`, de la siguiente manera:

```
clapfAssocList = [(clapfNodeId, "clapf.htm")]
main           = renderHD clapfDoc clapfAssocList
```

La función `main` anterior genera una página de WWW, cuyo código HTML reside en el archivo `clapf.htm`. La página puede verse en la Fig. 5.1. [1]

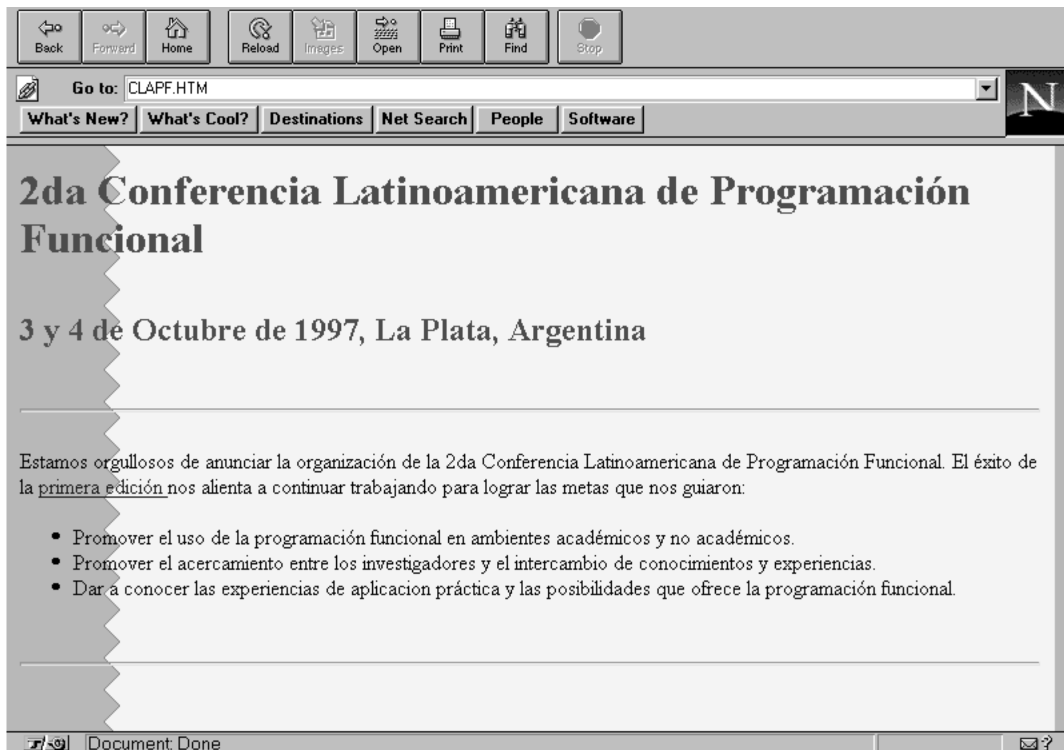

Figura 1: Página de la 2da Conferencia Latinoamericana de Programación Funcional.

[1] La visualización de las páginas HTML se realiza utilizando *Netscape Navigator Version 3.01*. Copyright 1994-1996 Netscape Communications Corporation.

5.2 Definiendo Nuevos Componentes

La motivación del segundo ejemplo es definir un documento con datos personales de dos investigadores. Por simplicidad, los datos que se consideran son nombre, fecha de nacimiento y posición dentro de la universidad. Además, se desea que estos datos se provean junto a una foto de la persona en cuestión.

Para dar una presentación estándar de los datos de las personas, y para evitar escribir código innecesario, se crea un componente reusable `personalFile`.

```
personalFile :: String -> String -> String -> ImgDumbHG -> OrgAnchorHG ->
                String -> DstAnchorHG
personalFile name date position image back id
              = dstAnchorHG id $ justifyC $
                (txtSize S6 $ txtColor Maroon $ txtDumbHG name) /=\
                vSpace 20                                       /=\
                image <=< dataList                              /=\
                back                                            /=\
                divisionLine
                where
                   item s1 s2 = (txtColor Green $ txtDumbHG s1) +++
                                (txtDumbHG s2)
                   dataList   = itemListDumbHG
                                    [item "Birth Date: " date,
                                     item "Position: " position]
```

El componente `personalFile` es un componente complejo, construído a partir de otros más simples; además, es un componente genérico que se instancia a partir de los datos de una persona en particular.

Además de cada una de las fichas personales, se desea tener un índice al principio de la página. Para esto, se construye el componente genérico `makeIndex`.

```
makeIndex    :: Hyperlink l => [(String, ImgDumbHG, l)] -> HG
makeIndex xs = justifyC $ foldr (<=<) nullHG (map build xs)
               where build (n,i,l) = justifyC $ (orgAnchorHG i l) /=\
                                                  (txtSize S4 $ txtDumbHG n)
```

La función `makeIndex` recibe una lista de tuplas, formadas por un `String`, un `ImgDumbHG`, y un elemento de la clase `Hyperlink`, que son el nombre, la imagen y el *link* a la ficha de esa persona, respectivamente. El resultado de la aplicación de la función es un componente complejo, en el que la imagen de una persona es *anchor* a su correspondiente ficha; además, debajo de cada imagen se coloca un componente de texto `TxtDumbHG` que corresponde al nombre de esa persona.

A continuación puede verse el código que describe el documento completo haciendo uso de los componentes genéricos `personalFile` y `makeIndex`.

```
import HTML     -- Modulo de la plataforma HTML.
import MAIN     -- Modulo principal.
import NEWC     -- Modulo con los componentes personalFile y makeIndex.

-- Definicion del titulo y el subtitulo.
title str     = justifyC $ txtLevel H1 $ txtColor Blue $ txtDumbHG str
titleId       = dstAnchorId "Title" thisNodeId
pageTitle str = dstAnchorHG titleId
                            (title "Functional Programming Researchers")

-- Definicion del indice y las secciones del documento.
imagePEML    = imgDumbHG (url "peml.jpg")
imageMAC     = imgDumbHG (url "mac.jpg")
namePEML     = "Pablo E. Martinez Lopez"
nameMAC      = "Marcelo A. Cardos"
pageIndex    = makeIndex [(namePEML, imagePEML, toSection1),
                          (nameMAC, imageMAC, toSection2)]
section1     = personalFile namePEML "20/3/1968" "Researcher"
                            imagePEML backToIndex
                            (dstAnchorId "PEML" thisNodeId)
section2     = personalFile nameMAC "4/4/1966" "Researcher"
                            imageMAC backToIndex
                            (dstAnchorId "MAC" thisNodeId)
backToIndex  = justifyC $ orgAnchorHG (txtDumbHG "Back to Index") toIndex

-- Definicion del cuerpo del documento.
pageBody     = pageTitle    /=\
               pageIndex    /=\
               divisionLine /=\
               section1     /=\
               section2

-- Definicion de hyperlinks.
toSection1   = nearHL section1
toSection2   = nearHL section2
toIndex      = nearHL pageTitle
```

```
-- Definicion del nodo.
thisNodeId   = nodeId "Second Example" thisDocId
thisNode     = backgroundImg (url "fondo2.gif") $ defaultTxtColor Black $
                    node thisNodeId pageBody

-- Definicion del documento.
thisDocId    = hdId "Second Example"
thisDoc      = hd thisDocId [thisNode]

-- Traduccion a la plataforma HTML.
assocList    = [(thisNodeId, "ejemplo2.htm")]
main         = renderHD thisDoc assocList
```

El documento, trasladado a plataforma HTML, puede verse en la Fig. 5.2.

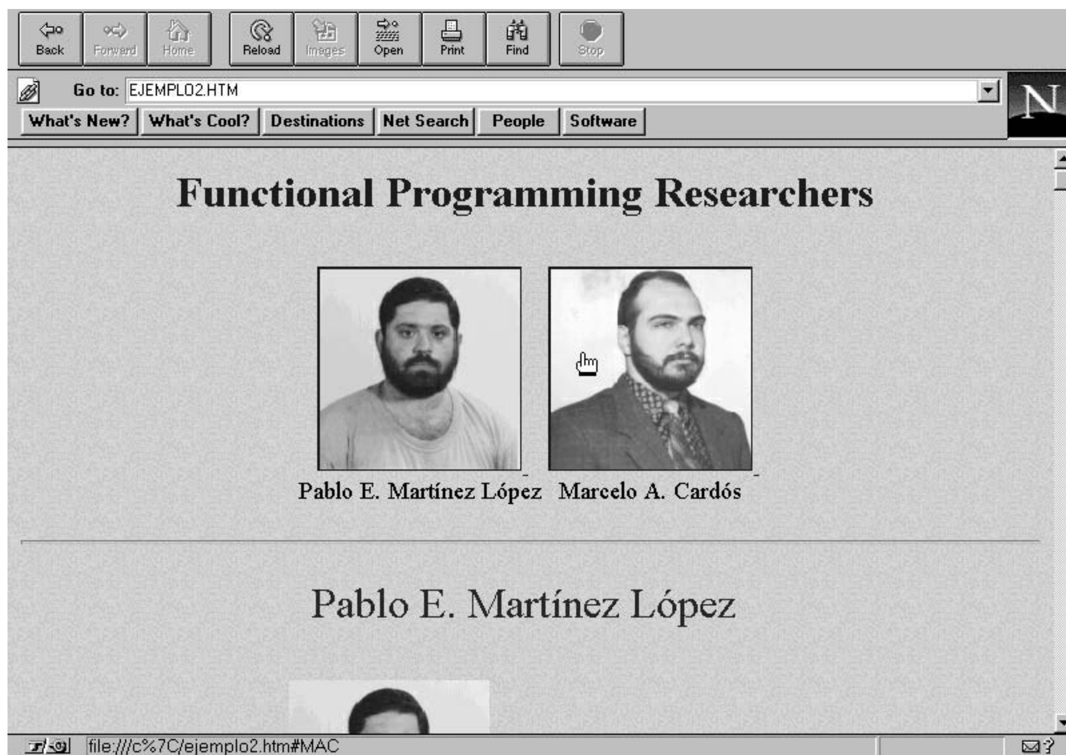

Figura 2: Página definida utilizando componentes genéricos.

5.3 Conectando los ejemplos anteriores

Supóngase ahora que se desea que el documento generado en el segundo ejemplo contenga un *link* al documento que se construyó en el primer ejemplo. Lo primero que debe hacerse es importar la definición del primer documento al archivo del segundo.

```
import CLAPF
```

El siguiente paso consiste en crear el *hyperlink*. Como se trata de documentos diferentes, el *link* a construir será de tipo `FarHL` (*Far Hyperlink*).

```
linkToCLaPF = farHL clapfNodeId
```

Una vez definido el *hyperlink* hay que incluírlo en el cuerpo del documento mediante el correspondiente *anchor*, modificando la definición de `pageBody` del ejemplo de la Secc. 5.2.

```
anchorToCLaPF = justifyC $
                orgAnchorHG (txtDumbHG "Go to ClaPF page") linkToCLaPF
pageBody      = pageTitle     /=\
                pageIndex     /=\
                divisionLine  /=\
                section1      /=\
                section2      /=\
                anchorToCLaPF
```

Una vez incluído en el cuerpo del documento, sólo resta transladarlo a la plataforma HTML. Como se tienen dos documentos, se utiliza la función `renderHDs` en vez de `renderHD`.

```
main = renderHDs [(clapfAssocList, clapfId),
                  (assocList, thisDocId)]
```

La página de WWW resultante puede verse en la Fig. 5.3.

6 Implementación

En esta sección se explican los puntos importantes sobre la implementación de algunas de las abstracciones mencionadas. Uno de los puntos a destacar es la forma en que se utilizaron las clases de tipos de Haskell para describir los conceptos de *hyperlink* e *Hypergadget*.

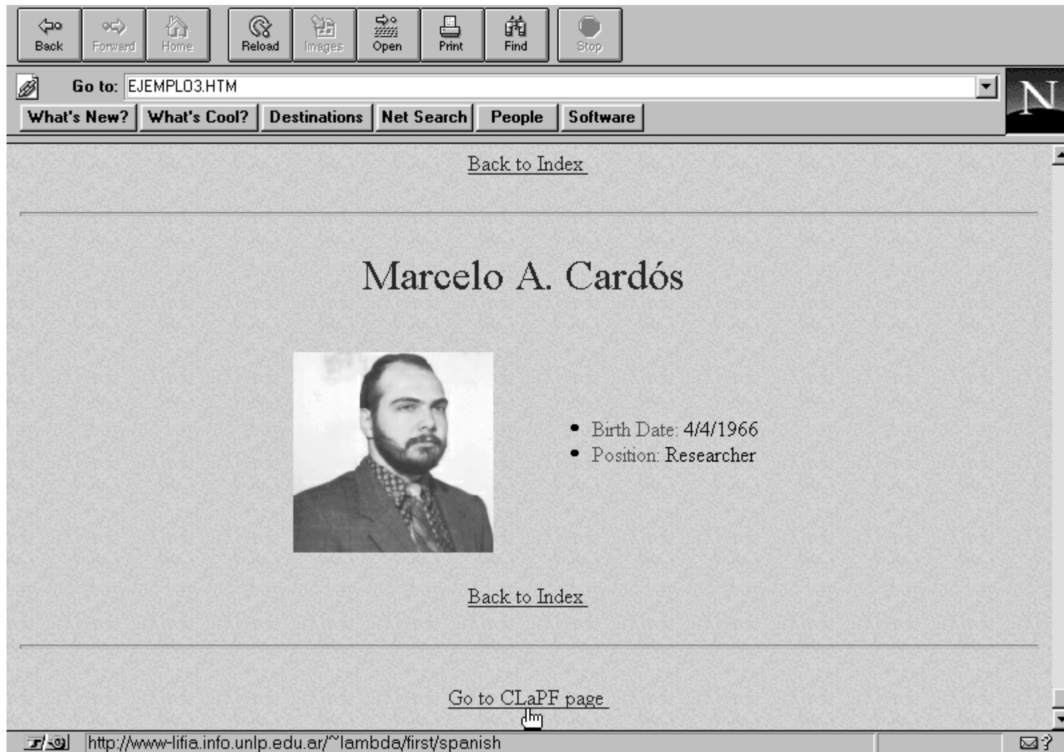

Figura 3: Extensión del segundo ejemplo.

6.1 Implementación de hyperlinks

Existen tres tipos que implementan el concepto de *hyperlink* dentro del trabajo. Estos tipos son `NearHL`, `FarHL` y `HL`.

Los tipos `NearHL` y `FarHL` representan los conceptos de *Near Hyperlink* y *Far Hyperlink*, respectivamente.

```
data NearHL = NearHL HLDestiny
data FarHL  = FarHL  HLDestiny
```

El tipo `HLDestiny` se detalla en la Secc. 6.1.1.

Un elemento del tipo `HL` encapsula, o bien a un elemento del tipo `NearHL`, o bien a uno del tipo `FarHL`.

```
data HL =   ANearHL NearHL
          | AFarHL FarHL
```

El tipo `HL` se utiliza junto con la clase `Hyperlink` para expresar *links* que pueden ser tanto *near* como *far*.

La clase `Hyperlink` agrupa a los tipos `HL`, `NearHL` y `FarHL`. Los elementos de estos tipos pueden convertirse al tipo general `HL`.

```
class Hyperlink h where
   toHL :: h -> HL
```

El tipo `HL` y la clase `Hyperlink` son utilizados por funciones polimórficas que toman o devuelven tanto *Near Hyperlinks* como *Far Hyperlinks*. Un ejemplo es la función `orgAnchorHG`, la cual construye un *anchor* de origen.

```
orgAnchorHG     :: (DumbHypergadget d, Hyperlink h) => d -> h -> OrgAnchorHG
orgAnchorHG d h = OrgAnchorHG (toDumbHG d) (toHL h)
```

La clase *DumbHypergadget* y la función `toDumbHG` se ven con detalle en la Secc. 6.2.

6.1.1 Hyperlink Destinys

El tipo `HLDestiny` (*Hyperlink Destiny*) describe el destino de un *hyperlink*.

```
data HLDestiny =   HLNodeDestiny NodeId
                 | HLAnchorDestiny DstAnchorId
                 | APSHLDestiny PSHLDestiny
```

El destino de un *hyperlink* puede ser un nodo, un *anchor*, o puede estar expresado por una construcción específica de alguna plataforma de *hypermedia* particular (por ejemplo, un URL en HTML). El tipo `PSHLDestiny` está implementado en el módulo de cada plataforma, y en particular en el módulo `HTML` está implementado como un sinónimo del tipo `URL`.

La clase `HyperlinkDestiny` agrupa a los tipos que pueden usarse para construir el destino de un *hyperlink*, y sus instancias son `Node`, `NodeId`, `DstAnchor`, `DstAnchorId`, y `PSHLDestiny`. Los elementos de los tipos mencionados pueden convertirse al tipo `HLDestiny`.

```
class HyperlinkDestiny d where
   toHLDestiny :: d -> HLDestiny
```

Un ejemplo típico del uso de esta clase es la función de construcción de *Near Hyperlinks*.

```
nearHL  :: HyperLinkDestiny d => d -> NearHL
nearHL d = NearHL (toHLDestiny d)
```

6.2 Implementación de Hypergadgets

El tipo más importante que representa a los *Hypergadgets* es el tipo `HG`.

```
data HG =   ADumbHG DumbHG
          | AnAnchorHG AnchorHG
          | AboveOf HG HG
```

```
            | LeftOf HG HG
            | JustifyL HG
            | JustifyC HG
            | JustifyR HG
            | NullHG
```

HG es el tipo de los *Hypergadgets* en forma general. Un elemento de este tipo puede encapsular a elementos de los tipos DumbHG y AnchorHG, y también a combinaciones y transformaciones de *Hypergadgets*.

La clase Hypergadget agrupa a los tipos que representan a los *Hypergadgets*. Los elementos de los tipos pertenecientes a esta clase pueden convertirse al tipo general HG.

```
class Hypergadget h where
   toHG :: h -> HG
```

Ejemplos de uso de esta clase son los combinadores y transformadores de *Hypergadgets*.

```
aboveOf     :: (Hypergadget a, Hypergadget b) => a -> b -> HG
above h1 h2 = (toHG h1) `AboveOf` (toHG h2)


justifyC    :: Hypergadget h => h -> HG
justifyC h  = JustifyC (toHG h)
```

De manera similar *DumbHG* es el tipo más general para expresar *Dumb Hypergadgets*.

```
data DumbHG =   ATxtDumbHG TxtDumbHG
              | AnImgDumbHG ImgDumbHG
              | AnItemListDumbHG ItemListDumbHG
              | DivisionLine
              | HSpacer Int
              | VSpacer Int
```

La clase DumbHypergadget agrupa a los tipos que denotan *Dumb Hypergadgets*, e incluye a TxtDumbHG, ImgDumbHG y ItemListDumbHG. Los elementos de los mencionados tipos pueden convertirse al tipo general DumbHG. Esta clase es subclase de Hypergadget, y por lo tanto sus instancias deben permitir también la conversión al tipo HG.

```
class Hypergadget h => DumbHypergadget h where
   toDumbHG :: h -> DumbHG
```

Un ejemplo del uso de esta clase es la función orgAnchorHG, vista en un ejemplo anterior (ver Secc. 6.1).

El tipo más general de los *Anchor Hypergadgets* es AnchorHG.

```
data AnchorHG =   AnOrgAnchorHG OrgAnchorHG
                | ADstAnchorHG  DstAnchorHG
```

La clase `AnchorHypergadget` agrupa a los tipos que denotan *Anchor Hypergadgets*, e incluye a `OrgAnchorHG` y `DstAnchorHG`. Los elementos de los mencionados tipos pueden convertirse al tipo general `AnchorHG`. Esta clase es subclase de `Hypergadget`, y por lo tanto sus instancias deben permitir también la conversión al tipo `HG`.

```
class Hypergadget h => AnchorHypergadget h where
   toAnchorHG :: h -> AnchorHG
```

Un ejemplo del uso de esta clase se da en la implementación de la función `toHG` para el tipo `DstAnchorHG`.

```
instance Hypergadget DstAnchorHG where
   toHG h = AnAnchorHG (toAnchorHG h)
```

7 Trabajos relacionados

La utilización de una *metodología composicional* para construir estructuras más complejas a partir de estructuras más simples no es nueva, y fue utilizada entre otros en [Fok95] para *parsers* funcionales, en [Hud96] para expresar composiciones musicales de forma declarativa, en [FJ96] y [HC95] para construir interfaces gráficas en lenguajes funcionales y en [FJ95] para expresar gráficos. En particular, la forma en que se utilizan transformadores y combinadores para expresar *layout* y detalles de presentación visual de componentes multimediales, es muy similar a la utilizada en [FJ96] y [HC95] para manejar el *layout* y presentación de componentes de GUIs y la utilizada en [FJ95] para definir la disposición física de componentes gráficos.

La construcción de estructuras de *hypermedia* genéricas y reusables y la creación de estructuras de *hypermedia* complejas a partir de la combinación de otras más simples ha sido utilizada, entre otros por [FNN96], desde la perspectiva del paradigma de orientación a objetos.

8 Conclusiones

Se ha presentado una forma de construir documentos *hypermediales* mediante herramientas construídas en un lenguaje funcional. Los beneficios usuales de la programación funcional se aplican a este trabajo: el alto nivel de abstracción, la gran capacidad de expresión y la facilidad de razonar sobre los programas, lograda gracias a la transparencia referencial.

El trabajo propone también el uso de combinadores y transformadores para construir estructuras complejas a partir de otras más simples. Esta característica, combinada con el uso de funciones de alto orden, permite la creación de componentes de *hypermedia* genéricos altamente reusables.

Por otro lado, las herramientas provistas permiten describir un documento *hypermedial* sin utilizar construcciones específicas de una plataforma de *hypermedia* particular, logrando que el documento pueda ser trasladado sin modificaciones a distintas plataformas.

Finalmente, otro punto a destacar de este trabajo es el acercamiento entre disciplinas habitualmente muy distanciadas en el ámbito informático latinoamericano, como son la construcción de sistemas de *hypermedia* y la programación funcional. Revisiones posteriores de las versiones preliminares de este trabajo por investigadores del área de *hypermedia* se han mostrado positivas respecto de la utilización del paradigma funcional en esta area. Lo anterior es un incentivo para continuar con el desarrollo de las ideas presentadas en este artículo.

Referencias

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[FJ95] Sigborn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Glasgow Functional Programming Workshop*, July 1995.

[FJ96] Sigborn Finne and Simon Peyton Jones. Composing the user interface with Haggis. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, LNCS 925*, pages 1–37. Springer-Verlag, Aug 1996.

[FNN96] S. Fraïssé, J. Nanard, and M. Nanard. Generating hypermedia from specifications by sketching multimedia templates. In *Proceedings of the ACM Multimedia '96*, 1996.

[Fok95] Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*, pages 1–23. Springer-Verlag, May 1995.

[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.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with fudgets. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*, pages 137–182. Springer-Verlag, May 1995.

[Hud96] Paul Hudak. Haskore music tutorial. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming, LNCS 925*, pages 38–67. Springer-Verlag, Aug 1996.

[Nie95] Jakob Nielsen. *Multimedia and Hypertext: The Internet and Beyond*. Academic Press Inc., 1995.

[PH+96] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale University, May 1996.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, LNCS 925*. Springer-Verlag, May 1995.