UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Proyecto de grado

EFECTIVIDAD DE LINKS-LANG

Efectividad del uso del lenguaje funcional Links
en la programación web.

AUTOR
Alejandro Schubert Bentancurt Sosa

SUPERVISORES
Alberto Pardo
Marcos Viera

Montevideo, Uruguay
2018

Resumen del trabajo

En la programación web actual, se puede ver la aparición de importantes cambios. En particular, es de destacar el nuevo impulso que ha tenido la programación funcional, especialmente en lo que refiere al front-end. En este trabajo nos concentramos en el análisis del lenguaje funcional Links, cuyo principal objetivo es utilizar un único lenguaje para el desarrollo de las tradicionales tres capas presentes en una aplicación web: presentación, lógica y persistencia. El objetivo de este trabajo es evaluar la efectividad de Links en la programación web actual mediante la construcción de un caso de estudio típico, aplicación web, y mediante casos de prueba sobre el propio lenguaje, y análisis de documentos disponibles.
Primeramente realizamos una introducción al lenguaje, mostrando sus aspectos más básicos y otros más elaborados que permitan entender el trabajo de evaluación realizado. A continuación nos introducimos en la construcción del caso de estudio, más el análisis del lenguaje en distintos aspectos relevantes para la programación web actual, además del análisis de otras características generales propias del propio lenguaje que serán de peso en su evaluación.
Llegaremos a la conclusión, que a pesar de grandes ventajas presentes, como la mucha mayor simplicidad del lenguaje para resolver problemas comunes, la factibilidad del lenguaje se ve comprometida por carencias en varios aspectos como seguridad y consultas a la base de datos.

Tabla de Contenido

1 Introducción

La programación actual está dominada por la programación imperativa. Esto es cierto en particular para el desarrollo web. Las aplicaciones web, representan actualmente un gran porcentaje del total de aplicaciones [5], dada la nula necesidad de instalación y su portabilidad de ejecución, desde distintos sistemas operativos hasta distintos dispositivos, como móviles y de escritorio.
Desde hace unos años, hay un crecimiento muy importante de la programación funcional, un paradigma de programación declarativa cuya base es la noción de función, y en su forma pura no hay estados. Así han aparecido nuevos lenguajes, como Reactjs, Scala, Elm, F#, se han agregado características funcionales a lenguajes más tradicionales, como Java (Java 8), y también han aparecido bibliotecas que siguen este paradigma, como Lodash en Javascript. El propio Javascript incorpora en buena medida conceptos de programación funcional, dado que las funciones son elementos de primera clase, se puede trabajar con ellas, pasarlas como parámetro a otras funciones, almacenarlas en variables, de igual manera que otro valor.
Dado este crecimiento del paradigma funcional, surgen preguntas como: ¿Es aplicable este paradigma al desarrollo web? ¿Usando solo programación funcional, es posible desarrollar una aplicación web de punta a punta? El objetivo de este trabajo es precisamente evaluar la efectividad de la programación funcional en el desarrollo e implementación de aplicaciones web; en particular este análisis se realizará focalizado en Links, un lenguaje con características funcionales [8]. Realizaremos este análisis mediante el desarrollo de una aplicación usando Links, la recopilación de información a este respecto desde la documentación disponible, y la creación de casos de prueba específicos para algunos problemas típicos. Interesará evaluar la legibilidad, mantenibilidad, correctitud, factibilidad y tiempo de desarrollo del posible producto.

La aplicación a desarrollarse en Links deberá ajustarse al objetivo del proyecto, tratando de explorar distintos aspectos de la programación web, como acceso a la base de datos, login, reportes, altas-bajas-modificaciones, relaciones entre entidades, acceso a web services, etc.
Se explorará la documentación prestando especial atención al objetivo, tratando de recuperar especialmente los aspectos relevantes para el mismo.
Los casos de prueba específicos ilustrarán situaciones no exploradas, o ejemplificarán aspectos destacados a exponer del lenguaje de uso relevante en el desarrollo web.

El presente informe se estructura en cuatro partes:

- una primera sección de estudio del lenguaje, análisis del mismo, estudio de toda la documentación asociada, creación de distintos casos de prueba, como acceso a base de datos, persistencia por el lado del cliente web, etc.
- una segunda sección con la creación de un producto web típico.
- una tercera parte de discusión y análisis de los distintos aspectos del lenguaje, junto a conclusiones particulares en estos aspectos.
- conclusiones generales

En las conclusiones finales abordamos la síntesis de todos los elementos analizados, tratando de llegar al objetivo final de evaluación desde un punto de vista global.

Previo a la introducción al lenguaje, y a la presentación del producto, parece importante establecer un breve contexto del trabajo, que es lo que se presenta a continuación.

1.1 La programación web tradicional

Para poder evaluar el lenguaje, es necesario establecer algún punto de comparación con respecto a la programación web tradicional. ¿Qué es la programación web tradicional? Para empezar, ¿qué lenguajes se utilizan?
Investigando el mercado de trabajo actual, y referencias como [5], puede decirse que las tecnologías más populares son: Java, SQL, Javascript. Además Hibernate es una muy popular herramienta de mapeo objeto-relacional (en inglés ORM). Por el lado del front-end puede agregarse que Angularjs [49] es un framework muy popular.

1.2 El paradigma funcional

Esta sección pretende mostrar los aspectos más relevantes de la programación funcional, a modo de contexto del lenguaje a evaluar. Por una introducción detallada a este paradigma puede consultarse [38].
La programación funcional se caracteriza por:

1. Noción de función. Es el primer aspecto esencial de la programación funcional. En el caso de la programación funcional pura a una entrada le corresponde una salida, donde no hay efectos, o sea no se hace algo más en el 'mundo' que devolver el resultado, no se cambian variables globales, ni otras modificaciones del 'mundo'.
2. Las funciones son "valores" de primera clase, no hay diferencia entre tratar un valor literal por ejemplo y una función. Por lo tanto las funciones se pueden pasar a otras funciones como parámetro. Esto permite definir funciones de alto orden, que son funciones que reciben funciones como parámetro o retornan funciones como resultado. Esta característica brinda gran potencia al lenguaje.
3. Inmutabilidad de los datos. No es posible modificar el valor de una variable, o sea, no existe el concepto de estado de una variable.
4. Tipado fuerte. Esta es otra característica común en muchos lenguajes de programación funcional. El dominio de las funciones está bien definido, explícita o implícitamente (a través de inferencia de tipos). Las funciones no se pueden aplicar a valores que no sean del tipo de su dominio.
5. Listas por comprensión. Casi una traslación de como matemáticamente se define un conjunto por comprensión, a la programación. Permite definir un conjunto (lista) mediante una notación matemática típica.
6. Demostración de correctitud del resultado. En programación funcional se puede demostrar que cierta función cumple ciertas propiedades, lo que no se puede hacer tan fácilmente en programación imperativa.

2 Antecedentes

El objetivo del proyecto es evaluar la efectividad del lenguaje Links para la creación de aplicaciones web, especialmente desde el punto de vista de la efectividad de la programación funcional.
Hay varios trabajos previos relacionados con el objetivo de este proyecto. Trabajos que aborden la creación de una aplicación web en Links aportan a este proyecto. Se puede obtener información sobre los problemas involucrados, ventajas que se tuvieron al crear el proyecto, comparaciones con otros lenguajes, etc.
Algunos trabajos son especialmente destacados, ya que se enfocan en la efectividad especialmente del lenguaje Links, como es el caso de [6]. Allí se crea una aplicación web y se trata de evaluar la capacidad de Links para el desarrollo web. Dicho trabajo tiene la desventaja de ser antiguo, y que han habido cambios en el lenguaje desde ese entonces. También se observa una muy cercana relación del autor con los desarrolladores del lenguaje. Esto puede llevar a perder cierta objetividad al realizar el trabajo, ya que muchas soluciones a problemas que surgen las brindan estos propios desarrolladores. De todas maneras este trabajo representa un gran aporte a este proyecto. Otro trabajo destacado es [7], donde se crea un IDE online para Links y se abordan problemas especialmente de front-end, aunque también varios de back-end. Las mismas observaciones anteriores son válidas para este trabajo. Todos los trabajos de evaluación directa del lenguaje ocurrieron hasta 2010, cuando finalizaba la primera etapa de desarrollo de Links [8], de los cuales hay dos disponibles [8] y se desarrollaron en estrecho contacto con el equipo de desarrollo original.
También hay otros trabajos que involucran la creación de aplicaciones en Links, como es el caso de [9]. A pesar de que el objetivo de dicho trabajo es la creación de un lenguaje que integre base de datos y la creación de páginas wiki, especialmente orientado al manejo de información científica, en el desarrollo y conclusiones del mismo se observan resultados que aportan mucho a este proyecto, por ser un trabajo realizado por los mismos desarrolladores de Links, y que es de gran volumen (más de 4500 líneas). Este trabajo aporta especialmente por el lado de performance, y ciertas limitaciones del lenguaje, como comunicación con otro software.
Existen también otros trabajos que aportan, explican y justifican nuevas funcionalidades del lenguaje como query shredding [10]. Estos trabajos aportan a este proyecto en sí mismos, al incluir nuevas características que pueden usarse en el desarrollo web.
En resumen se puede ver que hay varias fuentes que aportan a este proyecto.
Este proyecto pretende aportar un punto de vista más actual, a la vez de una visión integral de todos los aspectos del lenguaje involucrados en la creación de una aplicación web.

3 El lenguaje Links

Links es un lenguaje funcional, diseñado para su uso en la programación web. En la programación web, es necesario aprender muchos lenguajes para crear una sola aplicación, Javascript para el front-end, Java, PHP o .NET, etc, para la capa de servicios y SQL para alguna base de datos. El problema de la multiplicidad de lenguajes es conocido como

impedance mismatch problem [2], ya que es necesario crear distintas interfaces entre un lenguaje y otro, conversiones de tipos, etc. Links permite, mediante el uso de un único lenguaje, crear una aplicación para el cliente y el servidor, implementando las tres capas clásicas, de datos, servicios y presentación [2, 14]. Links es fuertemente tipado [2], usa inferencia de tipos Hindley-Milner [2], por lo que no es necesario declarar el tipo de cada función. El compilador realiza deducciones de cual tipo debería ser, y así se comprueban en cada expresión. Este compilador está escrito en O'Caml [2]. En cuanto a su ejecución, hay una primera fase de compilación a un lenguaje intermedio IR (*intermediate representation*) [17, 18, 19], que luego es interpretado. El intérprete se encarga de traducir las partes que corren en el servidor a código de máquina en el servidor, y las partes que corren en el cliente a Javascript [21]. A su vez, las partes que corren en el servidor se convierten en SQL en caso que sean consultas a una base de datos. Links solo corre en Linux, no en Windows. Su instalación se realiza a través de OPAM [35], un gestor de paquetes similar a Maven [36], donde se pueden instalar paquetes de O'Caml, en particular Links.
De las características anteriores, inmediatamente surgen algunas conclusiones y preguntas: los navegadores web solo entienden Html, Css y Javascript, por lo menos por ahora (https://webassembly.org/) ¿Links sustituye todos estos lenguajes?, ¿cuáles sí y cuáles no y en qué medida? Links soporta el tipo Xml y es posible crear Xml fácilmente. Como Html es 'en cierta medida' Xml, se puede escribir Html en Links (XHtml). No existe el tipo Css, pero sí se puede integrar de las tres maneras típicas posibles: inline, en cada elemento Html, de manera interna en el head y mediante un archivo exterior. Links deduce qué funciones se van a ejecutar en el cliente, y traduce estas funciones a Javascript. La interacción con el usuario se da por eventos en el cliente, los eventos tradicionales de los elementos Html, tienen su equivalencia en eventos Links, como 'onclick' por ejemplo.
También surgen las mismas preguntas por el lado del servidor, en la tradicional capa de persistencia. En la gran mayoría de aplicaciones se quiere persistir la información, esto va en contra de la programación funcional pura, ¿como se persiste la información? La información se persiste con una interfaz con efectos hacia una base de datos. Existen funciones que interactúan con la base de datos, permitiendo hacer consultas, insertar datos, y eliminarlos.
También todas las aplicaciones Links son escalables en cuanto al uso de memoria del servidor por ejemplo, ya que toda la información de estado se mantiene en el cliente como se explicará más adelante en la sección continuaciones.
Links tiene dos modos de ejecución, uno mediante un interactivo, que permite evaluar expresiones, y otro que permite ejecutar un archivo escrito en Links.

Ahora que tenemos un panorama general del lenguaje, vamos a entrar en más profundidad.

Con el objetivo de hacer el lenguaje más accesible a los desarrolladores web, a pesar de ser un lenguaje funcional, Links tiene sintaxis similar a Javascript [6, 7]. Tanto la definición de funciones, como la declaración de variables, son similares a Javascript. Lo mismo ocurre con las funciones de control, como if, switch, e incluso for. También posee las funciones de alto orden típicas de los lenguajes funcionales, como map, fold, etc. Links tiene esta forma para cumplir con el objetivo de ser adoptado fácilmente por desarrolladores web [6, 8].

Expresiones y secuencias

Las expresiones en Links pueden agruparse en secuencias mediante punto y coma. Una expresión cualquiera debe tener un resultado, éste está indicado por el último valor sin punto y coma. También las expresiones se pueden agrupar mediante llaves, por ejemplo:
{expr1;expr2;expr3}
es una expresión que ejecuta expr1, expr2 y devuelve expr3.

Con respecto a la evaluación de las expresiones, Links es un lenguaje estricto, su estrategia de evaluación es leftmost-innermost.
Por ejemplo no es posible definir como en Haskell:
links> fun ones() {[1] ++ ones()};
ones = fun : () ~> [Int]

y obtener:
links> hd(ones());

en este caso Links se queda en bucle infinito. No como en Haskell que devuelve 1.

Comentarios

Los comentarios solo pueden ser de una línea, y comienzan con "#".

Tipos básicos

Links posee los siguientes tipos básicos:

- *Int*, enteros
- *Float*, números de punto flotante: 3., 14.5
- *Bool*, booleanos
- *Char*, caracteres 'a', '9', '\012'.
- *String*

Es importante destacar que los Int se diferencian de los Float, por medio de un punto, el punto siempre acompaña al Float, incluso si no tiene fracción: 3. es un Float, 3 es un Int.
Hay funciones de conversión entre los tipos, por ejemplo *intToString* convierte a string un entero.
Son destacables las funciones *implode* y *explode*, que convierten strings hacia y desde lista de caracteres.

Operadores

Links posee los operadores:

- \+ : (Int, Int) -> Int
- \- : (Int, Int) -> Int
- * : (Int, Int) -> Int
- / : (Int, Int) -> Int

- ^ : (Int, Int) -> Int
- mod : (Int, Int) -> Int
- *. : (Float, Float) -> Float
- +. : (Float, Float) -> Float
- -. : (Float, Float) -> Float
- /. : (Float, Float) -> Float
- ^. : (Float, Float) -> Float

Acá se puede ver como se diferencia entre los operadores para Int y para Float. Links no soporta sobrecarga de operadores, así que deben llamarse distintas funciones si se usan tipos distintos. La operación de “2 + 2” no es la misma de “2. +. 2.”. Tampoco soporta *overloading* de funciones en general, solo puede definirse una sola función con el mismo nombre [6].

Listas

Links posee el tipo lista, donde todos los elementos tienen que ser del mismo tipo, por ejemplo:
[] # lista vacía
[1, 2, 3] # lista de enteros.
[1, 2.] # no es una lista

Operadores sobre listas son por ejemplo:
- ::, agrega un elemento al principio de una lista
- ++, concatena listas
- ==, compara listas

Funciones clásicas sobre listas:
- *hd : ([a]) -> a*, obtiene el primer elemento
- *take : (Int, [a]) -> [a]*, devuelve una lista con los primeros n elementos
- *drop : (Int, [a]) -> [a]*, devuelve una lista quitando los primeros n elementos
- *fold_left* y *fold_right*, funciones que permiten recorrer listas y operar en ellas.

Las funciones en Links no están currificadas.

Pattern matching en listas y constantes

Se pueden implementar funciones sobre listas por pattern matching utilizando el operador switch. Por ejemplo,

```
switch (s) {
  case []    -> “Empty”
  case x::xs -> “NonEmpty”
  other -> “Unreachable”
}
```

Si *s* es una lista, en caso de ser vacía se devolverá “Empty”; caso contrario, si la lista no es vacía se retorna “NonEmpty”.
La condición *other* se alcanza si no es verdadera ninguna de las anteriores, y siempre se ejecuta su sentencia asociada. En este caso eso nunca ocurre ya que siempre una de las primeras dos condiciones es verdadera.

Además de listas, también se puede realizar pattern matching sobre constantes, variantes y enumerados. El siguiente es un ejemplo con strings:

```
switch (myString) {
  case "Red" -> "Color rojo"
  case "Blue" -> "Color azul"
}
```

Aquí se produce un error si *myString* no es ni “*Red*” ni “*Blue*”.

También es posible el uso de pattern matching anidado:

case Node (Leaf, x, Node(_,y,_)) -> “A la izquierda es una hoja”

Rango de enteros

Se indican con [a..b]. Devuelve la lista de enteros entre a y b.

Tuplas y Records

Una tupla es un conjunto ordenado de valores, los cuales pueden tener tipos diferentes:

```
links> (1,'a',"");
(1, 'a', "") : (Int, Char, String)
```

Aquí el interactivo de Links (*links>*), muestra que el tipo del valor *(1,'a',"")*, es *(Int, Char, String).*
Podemos acceder a los elementos de la tupla con algunos operadores predefinidos como es el caso de *first*:

```
links> first((1,'a',""));
1 : Int
```

o también accediendo directamente al n-ésimo componente con el operador ‘.n’, por ejemplo:

```
links> (1,'a',"").2;
'a' : Char
```

Un record es una tupla, pero donde cada valor tiene un nombre, una etiqueta, por el cual se le puede hacer referencia:

links> (id=1,myChar='a',myString="");
(id = 1, myChar = 'a', myString = "") : (id:Int,myChar:Char,myString:String)

Un record no tiene los valores ordenados. Se puede hacer referencia a cada campo en el record usando el habitual operador de acceso (.). Si llamamos r al record anterior:

links> r.id;
1 : Int

Modificando Records

Se puede agregar y modificar campos a los records. Para agregar el campo *myField* al record *myRecord*:

(myField = "myValue" | myRecord);

donde *myRecord* es igual al record anteriormente usado.

Para modificar el campo myChar al record *myRecord*:

(myRecord with myChar = 'b')

Como mencionamos anteriormente, ninguno de los tipos es necesario declararlos, sino que se pueden usar directamente. Por ejemplo, se puede llamar a una función con el parámetro *(id=1, myField="algo")*.

Especificando el tipo de una expresión

Se puede especificar en forma explícita el tipo de cualquier expresión agregando ':' y después su tipo. Por ejemplo:

links> "":String;
"" : String

Enumerados

Cualquier literal que comience con mayúsculas se considera parte de un enumerado:

links> Red;
Red : [|Red|_::Any|]

Aquí se pide evaluar el tipo Red. La inferencia de tipos dice que es un enumerado, que contiene por lo menos el tipo Red, y además cualquier otro.

Valores de tipo enumerado se pueden evaluar con pattern matching:

fun evalMyEnum(val) {

```
  switch (val)  {
    case Red -> "Es rojo"
    case Blue -> "Es Azul"
  }
}
```

Análogamente a la misma función sobre *Strings* que se indicó más arriba, esta función en caso de recibir un valor distinto de *Red* o *Blue* produce un error, por lo tanto su tipo solo puede ser este:

evalMyEnum = fun : ([|Blue|Red|]) -> String

En cambio, si agregamos *other:*

```
fun evalMyEnum(val) {
  switch (val)  {
    case Red -> "Es rojo"
    case Blue -> "Es Azul"
    case other-> "Otro desconocido"
  }
}
```

ahora el tipo es:

evalMyEnum = fun : ([|Blue|Red|_|]) -> String

O sea, agrega el indicador "_" para decir que el enumerado puede contener cualquier otro valor.

Variantes polimórficas

Los anteriores enumerados pueden tener 'variantes', significando que pueden estar parametrizados por valores. Por ejemplo al tipo *Pantalon* se le quiere asociar un estilo:

```
links> Pantalon(Vestir);
Pantalon(Vestir) : [|Pantalon:[|Vestir|_::Any|]|_::Any|]
```

Esto permite asociarle un valor variable a cada valor del enumerado, permitiendo de esta forma una mayor flexibilidad.

A su vez también podemos hacer lo mismo con *Zapato(41)*, dónde 41 es la medida europea:

```
links> Zapato(41);
Zapato(41) : [|Zapato:Int|_::Any|]
```

Y podemos asociar los dos en un nuevo tipo, esta vez especificando los estilos posibles a *Vestir* o *Jean* para el *Pantalon* por ejemplo:

[| Pantalon : [| Jean | Vestir |] | Zapato : Int |]

De esta forma, este nuevo tipo, que podemos llamar *Ropa,* contiene subtipos, como *Pantalon* y *Zapato*.
Más en general, se puede agrupar los valores de un enumerado en subtipos.

Estos valores se pueden extraer por pattern matching, y devolver por ejemplo un talle:

```
switch (ropa) {
  case Pantalon(Jean) -> 45
  case Pantalon(Vestir) -> 42
  case Zapato(numero) -> numero
}
```

Operadores relacionales

Los operadores relacionales son:

- ==, igual
- <>, distinto
- <, menor que
- >, mayor que
- <=, menor o igual que
- >=, mayor o igual que

los que se pueden combinar mediante el uso de operadores lógicos:

- &&
- ||
- not

El operador *not* es prefijo:
links> not(1==2);
true : Bool

Es importante notar que estos operadores soportan overloading; por ejemplo, se usa el mismo símbolo de igualdad == para todos los tipos básicos, Int, Float, Listas, etc.

Expresiones condicionales

Se dispone del uso de if-then-else. En este caso, todas las opciones deben estar presentes. Esto es debido a que a diferencia del if-then-else de programación imperativa, que es una instrucción, en este caso es una expresión, por lo tanto retorna un valor.

```
if (cond)
  expr1
else
  expr2
```

donde *expr1* y *expr2* deben ser del mismo tipo, y *cond* de tipo Bool.

Variables

Las variables en el sentido que se utilizan en la programación imperativa no existen en un lenguaje funcional, ya que implican cambio de estado. Lo que se hace en Links, para facilitar su uso, es utilizar variables como si fueran definidas en un *let-in* en programación funcional. O sea, se reemplaza el nombre de la variable en las sentencias que sigan, en el mismo bloque, por su expresión asociada en la declaración de variable var.
Así por ejemplo:

```
var x= "text";
x
```

devolverá el valor *“text”.*

El uso conjunto de if-then-else más el uso de variables se puede ver en este ejemplo:

```
  var x = if (condition)
    2
   else
    3;
  x
```

Luego se sustituye esta expresión completa en la x, y se devuelve entonces esta evaluación.
Al programar es importante tener en mente el verdadero significado de var, para no confundirlo con el habitual en programación imperativa.

En la redeclaración de variables, siempre se toma la última:

```
var x = 2;
var x= "text";
x
```

devuelve *“text”.*

Funciones

Como se pudo ver previamente, las funciones se pueden definir mediante la sintaxis:

```
fun myName (param1, param2, ..)
{
 body
}
```

también pueden ser anónimas:

```
fun (param1, param2, ..)
{
 body
}
```

Las funciones anónimas son expresiones que devuelven una función, y por lo tanto pueden ser usadas en variables:
var myFun = fun (x) {x+1};

myFun(2) devuelve 3.
También como se ve las funciones son prefijas.

Xml Quasiquotes

Links permite embeber expresiones Xml en cualquier programa, Xml es un tipo más, sobre el que se pueden establecer funciones por ejemplo:

links> <myTag>Contenido ..</myTag>;
<myTag>Contenido ..</myTag> : Xml

el tipo de *<myTag>Contenido ..</myTag>* es Xml.
También se pueden agrupar varios tags juntos con el tag <#>:

links> <#><a/></#>;
<a/> : Xml

esto permite manejar Html:

```
<html>
  <body>
    <ul>
      <li> item 1</li>
      <li> item 2 </li>
    </ul>
  </body>
</html>
```

Como se puede ver tiene que ser Html estricto, todos los tags deben cerrarse.
También se pueden embeber, dentro de Xml, expresiones de Links, mediante el uso de '{' y '}':

links> <#>Nombre: {stringToXml("miNombre")}</#>;
Nombre: miNombre : Xml

Aquí, dentro del tag 'b' se agrega un Xml, desde el string "miNombre".
Se puede escapar la llave con doble llave:

links> <#>Nombre: {{stringToXml("miNombre")}}</#>;
Nombre: {stringToXml("miNombre")} : Xml

Páginas

El tipo Xml no representa una página que se puede servir directamente vía servidor web. Para esto existe el tipo *page* que contiene al tipo Xml:

```
page
  <html>
    …
  </html>
```

Un valor de este tipo se puede expresar en Links para ser mostrado como página web, como veremos más adelante.

Forms

Para procesar información, Links solo soporta los elementos Html form. En los mismos se puede indicar cuales son los campos a procesar, y de qué manera se realizará este procesamiento, si se hace un POST al servidor, o se hace por AJAX. Los campos a procesar, representados por el elemento *input,* siempre se indican con el atributo *l:name*, que sustituye al *name*. Por ejemplo:

```
<form>
  <input type=”text” l:name=”apellido” />
</form>
```

es un form que cargará el valor del campo con *l:name* = *“apellido”*, en la variable *apellido*. La variable se procesa en la cabecera del form:

```
<form l:action=”{myFun(apellido)}”>
  <input type=”text” l:name=”apellido” />
  <input type=”submit”/>
</form>
```

aquí desde el *l:action* del form, se puede acceder al campo *apellido*. *l:action* hace un “submit” al servidor, envía la información y espera una nueva página web. Esta nueva página web es lo que devuelve la expresión entre llaves de *l:action*, debe ser de tipo page, página, y reemplaza la página actual.

Así, sí *myFun* es:

```
fun myFun(apellido) {
  page
    <html>
      <p>Tu apellido es: {stringToXml(apellido)}</p>
    </html>
}
```

Se mostrará una página con el apellido ingresado si se hace submit.

La otra posibilidad, en vez de usar *l:action* en el form, es *l:onsubmit*. Esta función no hace un submit, sino solo procesa su contenido. Por ejemplo:

```
<form l:onsubmit="{print(apellido)}">
  <input type="text" l:name="apellido" />
  <input type="submit"/>
</form>
```

Solo mostrará el apellido en la salida estándar del servidor, no cambiará la página actual. Esto es útil para utilizarlo junto con AJAX, es útil para modificar la página actual con valores del *form* o del servidor.

Componentización - Abstracción de páginas

Los atributos de los *forms* tienen ciertas incompatibilidades con la abstracción de páginas. Se podría pensar en construir funciones que devuelvan componentes, segmentos de Xml, y tengan incrustados atributos *l:xx*. De esta manera se pueden crear componentes como radiobuttons, datepickers, etc para ser reutilizados. Lamentablemente esto no es totalmente compatible con Xml y Page. Pero sí se puede hacer con page splice [30]:

```
<main>
  {| content |}
</main>
```

Como se puede observar, la manera de incluir *content* es mediante los símbolos *{| … |}*, no mediante *{ ... }* como se indicó previamente. En este caso sí se preserva el funcionamiento con los *l:name*, y por lo tanto se pueden usar. El tipo de content debe ser *Page*, no Xml.

***For*, listas por comprensión**

Como se indicó previamente, Links incluye una primitiva *for* con la idea de hacer más cercano el lenguaje a los desarrolladores, más parecido a la programación imperativa estándar. Veamos un ejemplo:

```
for (x <- [1,2,3])
    [x*2]
```

devuelve [2, 4, 6]. Esto equivale a una lista por comprensión, como ocurre en Haskell:
*{x*2 | x <- [1,2,3]}*

Más en general en:

```
for (x <- list)
    resultList
```

list y resultList son expresiones que evalúan a listas. *resultList* es una expresión que depende de x, una función que dado un x, devuelve una lista. Luego estas listas son todas concatenadas, como pudo verse en el ejemplo anterior.

Links provee la capacidad de filtrar resultados con *where*:

```
for (x <- [1,2,3])
  where (x==2)
    [x*2]
```

Este código en listas por comprensión sería: {x*2 | x <- [1,2,3], x == 2}, lo cual evalúa a [4]. No se incluyen los resultados que no cumplan con la condición asociada al *where*.

```
where (cond)
expr
```

es equivalente a:

```
if (cond)
  expr
else []
```

También se puede ordenar en el *for*, con la cláusula *orderby*:

```
 for (x <- [(id=1,name="name1"),(id=2,name="name2")])
  orderby (x.id)
    [x]
```

Se ordena en forma ascendente.

For también soporta anidación. Como la expresión de salida del *for* es una lista, dicha expresión puede ser a su vez otro for:

```
for (x <- [1,2,3])
  for (y <- [4,5,6])
    [(x,y)]
```

devuelve *[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)].*

También se soporta Xml como tipo resultado, lo que es muy útil aplicado a páginas, por ejemplo tablas:

```
for (d <-[1,2,3,4])
  <tr><td>El valor es: {stringToXml(d)}</td></tr>
```

El anterior código devuelve 4 filas de *tr,* concatena los Xml, a pesar que no son una lista. Por lo tanto el *for* también soporta el tipo Xml.

3.1 Acceso a la Base de Datos

El acceso a la base de datos fue originalmente inspirado en Kleisli [6]. La solución convencional de permitir acceso a la base de datos es mediante conexiones y luego realizar consultas sobre esas conexiones, en forma de strings SQL. Estos string no tienen ningún chequeo de tipo ni de sintaxis, simplemente se envían a la base de datos y se espera una respuesta. Estas consultas además pueden tener ciertos valores del lenguaje de programación embebidos, como condiciones en la misma. El resultado devuelto debe convertirse nuevamente a los tipos del lenguaje anfitrión.
Links no utiliza este enfoque, sino que integra, sin agregar expresiones SQL, consultas a la base de datos. Se utiliza para ello la misma sintaxis y los tipos que provee el lenguaje. Existen funciones que devuelven bases de datos y tablas.

Para conectarse a una base de datos se escribe:
var db = database "myDB" "postgresql" ":5432:user:userPass";

De esta forma se conecta a la BD *myDB* con el driver *postgresql* en el puerto *5432* con credenciales *user/userPass.*
Para esto es necesario haber agregado este driver en la instalación de Links, esto puede verse en detalle en el Anexo A.

Para obtener un manejador de una tabla se usa:

var myTable = table "cursos" with (id : Int, nombre : String) from db

donde se obtiene un manejador de la tabla *cursos*, tal que sus campos son mapeados a los tipos indicados, desde la base de datos *db*. Aquí los tipos son los tipos del lenguaje.

Ahora se pueden hacer consultas sobre la tabla, con el *for* explicado anteriormente para listas por comprensión:

```
for (item <-- myTable )
 [item]
```

Esto devolverá toda la tabla *myTable*, en el caso del ejemplo anterior serán pares (id, nombre).
Como se puede ver, hay una diferencia con el formato anterior del *for*. Ahora se usa '<--' en vez de '<-' en la cláusula del *for*. Esto es debido a que ahora se usan manejadores de tablas, no listas como anteriormente.
La consulta:

```
var myTable = table "cursos" with (id : Int, nombre : String) from db;
for (item <-- myTable )
 [item]
```

Se convierte a una consulta SQL contra la BDs:

```
SELECT *
FROM cursos;
```

Esta es una consulta muy simple, quizás la más simple que se puede hacer. Links no solo permite convertir este tipo de segmentos de código Links a SQL, sino también más complejos, usando las cláusulas auxiliares del for que vimos antes:

```
var myTable = table "cursos" with (id : Int, nombre : String) from db;
for (item <-- myTable )
  where (item.id  <> 1)
  orderby (item.nombre)
 [item]
```

Se convierte en:

```
SELECT *
FROM cursos
WHERE id <> 1
ORDER BY nombre;
```

También se pueden usar funciones, como por ejemplo una función que devuelva la tabla *myTable* anterior, en vez de usar una variable.

Esto no siempre se puede hacer, no todos los fragmentos *for* se pueden convertir a código SQL. Al usar la cláusula <--, se indica a Links que el programador desea convertir todo el código Links del *for* a SQL. Si esto no es posible Links mostrará un error. Más información puede verse más adelante en Abstracción de la Base de Datos.

Si no puede convertirse todo el código del *for* a una única consulta SQL, puede quererse iterar sobre el contenido completo de la tabla. Para esto existe la función *asList*, que dado un manejador de tablas, como el *myTable* anterior, lo convierte en una lista. Así se puede usar esta función para extraer la lista, y pasarla al *for*:

```
for (item <- asList(myTable) )
  where (item.id  <> 1)
  orderby (item.nombre)
 [item]
```

El resultado es el mismo, pero la forma de implementación es muy diferente. En este caso se hace primero un SELECT * FROM cursos, y esto se convierte a una lista mediante *asList*, que lo pasa al for. O sea todo el filtrado no lo hace la BDs, sino Links programáticamente, en memoria.

También es posible tomar solamente los *n* primeros resultados, con la función *take*, que se traduce a *limit* de SQL. Además se puede usar *drop* junto con *take*, para agregar un *offset* [6]:

```
take (n, drop (m, for ...))
```

equivale a:

```
SELECT …
FROM …
WHERE …
LIMIT n OFFSET m
```

También se traducen *fors* anidados[9]**:**

```
for(x <-- tablaUno)
  for(y <--tablaDos)
    where (x.a == y.c)
    [(a=x.a, d=y.d)]
```

se traduce en:
SELECT x.a, y.d FROM tablaUno x, tablaDos y WHERE x.a = y.c

Bloque "query"

Se puede chequear que cierta expresión sea trasladable enteramente a SQL. Para esto se usa la palabra reservada query:
query {
expr
}
si *expr* es *wild* (ver la sección abstracción de la base de datos) producirá un error, sino continuará.

Modificaciones en la Base de Datos

Links además de consultas para solo extraer información, también permite modificarla en la base de datos. Permite ejecutar insert, update y delete.

Una vez definida la tabla, se puede insertar valores a la misma mediante:
insert myTableCursos values (id,nombre) [(i = 2, nombre = "admin")]

También se puede actualizar información:
update (r <-- myTableCursos)
where (r.id == 1)
set (i = 1, nombre = "admin2")

y borrar una fila:
delete (r <-- *myTableCursos*)
where (r.id == 1)

Las cláusulas *where* son iguales a las del fo*r.*

Restricciones en las tablas

Links también soporta un tipado más detallado de las columnas de las tablas, inspirado en [11]. En general, las columnas de tablas en SQL se caracterizan por no solo poseer el tipo del dato que almacenan, como ser Integer, Bigint, Varchar, etc. sino también otras características, como valores por defecto, si pueden tomar valores nulos, largo máximo, etc. Links incorpora parte de esta información a los manejadores de tablas.
Si una columna puede tomar un valor por defecto, no es necesario agregarla al hacer un *insert.* Esto se indica al declarar la tabla en Links:
table "cursos" with (id : Int, nombre : String) where id default from db;

También cada campo podría ser de sólo lectura, u obligatorio (*needed*). En ese caso se debe cambiar *default* por *readonly.*
table "cursos" with (id : Int, nombre : String) where id readonly from db;

Si no se indica otra cosa, los campos son necesarios, deben ser incluidos para poder realizar la modificación a la BDs, y también no son readonly. Si un campo es *default*, no es necesario, se puede obviar al hacer un insert, y si es readonly, se debe obviar.
No existen otro tipo de restricciones en Links.

Abstracción de la Base de Datos

Como se mencionó previamente para el *for*, en el mismo se pueden usar ciertas funciones, como por ejemplo una que devuelva una tabla, y se le puede pedir a Links que ese *for* se convierta completamente en SQL. Links provee en el tipo de cada función un indicador que muestra si la función puede o no ser convertida a SQL. Las funciones que sí pueden ser convertidas, utilizan la flecha simple, ->, son llamadas *tame,* las que no pueden ser usadas, utilizan la flecha curvada, ~>, son llamadas *wild*. Así viendo su tipo, es posible saber cuáles se pueden usar y cuales no. Las funciones que usan funciones primitivas o son recursivas no se pueden usar, son *wild*.
Por ejemplo *implode* no se puede usar:
links> implode;
implode : ([Char]) ~> String

Cualquier código for con <-- que use esta función dará un error de compilación.

3.2 Expresiones regulares

Links también tiene soporte para expresiones regulares. El formato es:
myString =~ regExpr

donde *myString* es un *String* y *regExpr* una expresión regular rodeada de barras (/). Por ejemplo:
“myString” *=~ /m(y|a).*/*

devuelve true. Los operadores son los comunes en expresiones regulares, no es posible realizar sustitución.

Expresiones regulares y Base de Datos

Se pueden usar expresiones regulares simples con la cláusula *where* en consultas a la base de datos, para ser convertidas a condiciones con el operador *LIKE* de *SQL*. Esto solo ocurre en casos simples, como por ejemplo:
*where (item.nombre =~ /.*programaci.*/)*

3.3 Sistema de tipos

Links es lenguaje fuertemente tipado y de tipado estático.
Por lo tanto cada función acepta solo ciertos tipos, y en caso de pasarse como argumento valores que no son del tipo de la función, se reportará un error.

Previamente se comentaron los tipos básicos, Int, Bool, Float, String, y Xml. El sistema de tipos en Links es complejo, y utiliza inferencia de tipos, *row polymorphism* y efectos. Cada uno se explicará a continuación.

Inferencia de tipos

Links utiliza inferencia de tipos. No es necesario declarar el tipo de un elemento, el sistema deduce de que tipo debe ser.
Por ejemplo:
```
fun myFun(x) {
  x+2
}
```
es de tipo:
```
fun : (Int) -> Int
```
2 es de tipo Int, x se suma a 2, como el + tiene dominio y codominio Int, x debe ser de tipo Int, además, el resultado de myFun es el resultado del +, entonces es de tipo Int.

También puede ocurrir que haya funciones que tengan esta forma:
```
links> fun myFun(r) {r.id};
myFun = fun : ((id:a::Any|_)) -> a::Any
```

r es un record, que contiene el campo id, pero no conocemos el tipo del campo id, podría ser cualquiera. Así aparecen variables de tipos, en este caso *a.* La variable *a* representa el tipo de r.id. Como no se conoce su valor, se representa mediante una variable.

Se puede observar algo más, que agrega complejidad: *a::Any.* Es un tipo sobre la variable de tipo *a*, un super tipo. La variable de tipo *a*, puede ser de tipo *Any* o tipo *Base*. El super tipo *Any* indica que puede ser de cualquier tipo, en cambio *Base*, que es de un tipo básico, Int, Char, String, Float. Estos super tipos fueron introducidos para ayudar al compilador a resolver si una expresión puede ser convertida a SQL [1].

Polimorfismo de filas

Como se explicó previamente para el caso de tipos de expresiones con switch, los tipos inferidos por Links utilizan todos los valores deducidos más una valor genérico.
Por ejemplo:
```
links> Red;
Red : [|Red|_::Any|]
```

Aquí el tipo de *Red*, no es solo el enumerado *Red*, *[|Red|]*, como podría pensarse a primera vista, sino además, hay una variable genérica '_' que extiende el enumerado. En este caso los valores inferidos directamente son solo *Red*, pero el tipo más general podría contener otros valores. De esta manera se obtiene mediante el agregado de una variable genérica, el tipo más general posible, un enumerado que contenga a *Red*.

En el caso de funciones, como por ejemplo:
```
links> fun myFun(r) {r.id+2};
```

myFun = fun : ((id:Int|_)) -> Int

ocurre algo similar, el tipo de la función no es un *record* que solo tiene el campo *id ((id : Int)),* sino un record que tiene el campo id, y además cualquier otro campo (distinto de id). De esta manera nuevamente se permite obtener el tipo más general posible de una función, o expresión. Esto se conoce como *row polymorphism* [16, 23]: se considera como tipo el conjunto de todos los conjuntos que contienen todos los valores deducidos. La extensión se representa mediante una variable extra que indica esta situación.
Entender esto es el primer paso para poder interpretar los tipos devueltos por Links, y poder firmar expresiones y funciones.

Las funciones en Links aceptan este tipo de polimorfismo, sin embargo, no soportan sobrecarga. Una función que acepta Int no puede aceptar Float. El row polymorphism se aplica a variantes, records y a efectos [24].

Firmas de funciones

Se puede indicar explícitamente el tipo de un función, mediante *sig*:

sig myFun : ((id:Int)) -> Int
fun myFun(r) {r.id+2};

links> myFun;
myFun = fun : ((id:Int)) -> Int

Como se puede apreciar, aquí el tipo es *((id:Int)) -> Int,* y no *((id:Int|_)) -> Int.* Mediante la firma, se restringió el dominio de la función. Esto se puede hacer en general, y también se pueden usar alias en las firmas de las funciones.

Alias de tipos

Con tipos muy complejos, es conveniente encontrar una manera de nombrar el tipo construido. Esto se puede hacer con:
typename myTypeName = type

por ejemplo:
typename Colors = [|Red|Blue|Yellow|Magenta|Green|];

Ahora se pueden especificar el tipo de una función o expresión mediante este alias:
links> Red : Colors;
Red : Colors

Tipos recursivos

Links también permite recursión de tipos:
typename List(a) = [| Nil : ()| Cons : (a, List(a)) |];

Así, se puede especificar un tipo a un valor:

links> Cons((1,Nil)) : List(Int);
Cons(1, Nil) : List (Int)

La representación de los tipos recursivos en Links es en términos del operador de punto fijo *mu* [34]:
links> typename List(a) = [| Nil| Cons : (a, List(a)) |];
List = a.mu b . [|Cons:(a, b)|Nil|]

Efectos

Los efectos agregan complejidad al sistema de tipos.
En programación funcional pura, una función para cada entrada tiene una salida, y no hace otra cosa. Las funciones son determinísticas, en el sentido de que para cada entrada siempre retornan la misma salida (el mismo resultado) y no dependen de en qué contexto se llaman. Pero cuando aparece "memoria externa" por ejemplo, es necesario que esa función además modifique el 'mundo exterior', así que además de retornar la salida habitual, puede que haga algo más. Ese algo más es un *efecto*. A su vez, estos efectos pueden tener tipos, puede haber tipos de efectos. Un efecto puede ser solo un marcador sobre una función, que indica que tipo de efectos puede hacer, o sea, que cosas interesa que pueda hacer. Un ejemplo previamente explicado son las funciones *tame* (no *wild*). Interesa saber si cierta función se puede convertir o no a SQL [22]. En este caso se marcan las funciones de esta manera, indicando que se puede usar. Es una clasificación extra de la función, en principio independiente del tipo de su entrada y salida, o sea al final, otra parte más del tipo de la función.
En resumen, los efectos de las funciones también tienen tipos, y al final el tipo de la función debe hacer referencia al tipo del efecto.
En Links hay dos tipos de efectos: el de las funciones *wild* y las que se usan en concurrencia, *hear:A*, que escuchan por eventos de tipo *A*. Las *wild* como se comentó usan ~>; en cambio las que escuchan usan *{hear:A}~>*.

También puede haber variables que representen efectos. Por ejemplo, en la función *all* que evalúa si todos los elementos de una lista cumplen una determinada propiedad:
links> all;
fun : ((a::Any) ~b~> Bool, [a::Any]) ~b~> Bool

El tipo de la función tiene como dominio una tupla, cuyo primer elemento es otra función que tiene un efecto *b,* y como segundo elemento una lista de tipo *a,* que puede ser de cualquier tipo y retorna como resultado, un valor de tipo Bool, pero también un efecto de tipo *b*. Esto es razonable, ya que la función *all* debería tener el mismo tipo de efecto sobre el mundo que el tipo de efecto del predicado al cual se le aplica. Además *all* es *wild*, no se puede convertir a SQL. Esto último actúa como marcador para saber que no se puede usar.
Es importante tener una idea de esta representación de los efectos, para poder entender el sistema de tipos, por ejemplo, cuando Links muestra el tipo de una función o expresión.

3.4 Componentización - formlets

Como se comentó previamente en abstracciones de páginas, la programación funcional, y Links en sí mismo, a través de la inclusión de Xml como tipo básico y embebido en el lenguaje, permite encapsular partes de Xml, para buscar su reutilización en otra parte, otro segmento de página Html, que lo contenga.
Links además provee un mecanismo para lograr esta componentización en *forms* de manera más directa, creando componentes en Links mediantes los llamados *formlets.* Estos permiten abstraer elementos solo dentro de una *form*.
Un ejemplo típico de componente es un *datepicker*, un fragmento de código que permita ingresar una fecha:

```
fun dateFormlet() {
  formlet <#>
    <span>Fecha:</span> <br />
    Day:   {inputInt -> day}
    Month: {inputInt -> month}
    Year:  {inputInt -> year}
  </#>
  yields {
    Date(day, month, year)
  }
}
```

Este *formlet* permite ingresar una fecha, mediante 3 enteros, día, mes, y año.
Además devuelve, previo al envío del formulario (*submit form*), un variante, que tiene como parámetros el día, mes y año.
Hay dos palabras clave, *formlet* y *yields*. Entre las dos está el código Xml que se va a mostrar, y después de *yields*, lo que devuelve el componente, en este caso la fecha.

Para procesar el valor devuelto por un formet es necesario un manejador, que es una función que recibe como parámetro el valor de salida del *formlet*, y devuelve una nueva página a mostrar, al haber hecho *submit* el usuario.
Se representa por:

```
{dateFormlet() => miManejador}
```

Siendo miManejador:

```
fun miManejador(Date(d,m,a)) {
 page
  <html>
   <body>
    <h3>La fecha es:</h3>
    <p>{intToXml(d)}/{intToXml(m)}/{intToXml(a)}</p>
    {dateFormlet() => miManejador}
   </body>
  </html>
```

}

A su vez volvimos a incluir el formlet en la página devuelta.
Como se puede apreciar, este formlet a su vez utiliza otros *formlets*, los *inputInt*. Estos componentes los provee el lenguaje. Los *inputInt* consisten en elementos Html *input,* más ciertas validaciones, para permitir ingresar solamente enteros.

3.5 Concurrencia

La concurrencia es muy importante con el uso de interfaces de usuario, se pueden apreciar muy comúnmente animaciones indicando que la página se está cargando, etc. Es necesario no detener el thread principal de ejecución del programa, ya que podría congelarse la interfaz gráfica, al no poder responder a una interacción con el usuario, si este mismo thread está realizando otra tarea más pesada.
La concurrencia en Links está inspirada en Erlang [2, 9].
Se pueden crear nuevos procesos con la primitiva *spawn*:

```
var myProcId = spawn {
  myExpression
}
```

De esta manera se crea un nuevo proceso que ejecuta la expresión *myExpression*. Una vez que esta expresión termina, el proceso también lo hace.
spawn devuelve el id del proceso creado. Se puede conocer el id del proceso actual mediante *self()*.
La comunicación entre procesos se hace a través de *mailBoxes,* se pueden enviar mensajes a los procesos mediante la primitiva *!:*

```
value ! myProcId
```

A su vez en el proceso myProcId, se puede recibir el mensaje con:

```
receive {
   case pattter1 -> expression1
   case pattter2 -> expression2
}
```

Se hace *pattern matching* sobre el valor recibido, de igual manera a como se explicó con *switch*.

Existe cierta asimetría entre crear procesos en el servidor y crearlos en el cliente. La anterior primitiva, *spawn* es para el servidor, para el cliente debe usarse *spawnClient* [30].
Se puede indicar que una función sea específicamente del servidor o del cliente, como veremos a continuación.

Código localizado

Dado un segmento de código, este corre en el cliente o en el servidor. Una función se puede invocar desde cualquiera de los dos, pero correrá en una u otra máquina. No todas

las funciones pueden correr en cualquier máquina, si realiza operaciones sobre la página web actual, como veremos más adelante, necesariamente esas operaciones deberán correr en el cliente. Por otro lado, si tiene acceso a la base de datos, deberá correr en el servidor, por ejemplo. Links puede inferir si una función corre en el servidor o en el cliente. La invocación de una función en el servidor desde el cliente, implica el uso de continuaciones y AJAX [6]. No siempre se quiere que esto ocurra, puede suceder que se prefiera que ciertas funciones se ejecuten sólo en el cliente, y se traduzcan a Javascript.
Links provee una manera de indicar explícitamente que una función debe correr en el servidor, o sea interpretarla desde el IR al servidor, o debe correr en el cliente, o sea debe compilarse a Javascript. Las primitivas son server y client, se colocan al definir la función justo antes de abrir las llaves:

```
fun serverFun server {
  ...
}

fun clientFun client {
  ...
}
```

3.6 Acciones del usuario

Una vez servidas las páginas web, el programa debe tener un punto de partida. Estas son las acciones que el usuario puede realizar con la página web. Por ejemplo puede hacer clic en un botón, rellenar un campo, pasar el ratón por encima de un componente, etc.
Estas acciones están determinadas por lo que puede entender el navegador: Html. Las acciones del usuario llegan al navegador, que las traslada al código Html de la página. Y el navegador está diseñado para poder detectar las acciones que pueden ser entendidas, enviadas a código Html. Por lo tanto debería haber cierta correspondencia entre los eventos Html y los de Links. Lo que no puede detectar Html no puede detectarlo Links.
Los eventos disponibles son:

- *l:onmousedown*
- *l:onmouseup*
- *l:onmousemove*
- *l:onmouseout*
- *l:onmouseenter*
- *l:onkeyup*
- *l:onkeydown*
- *l:onclick*
- *l:onfocus*
- *l:onchange*
- *l:onload*

Todos comienzan con *l:* al igual que los atributos previamente analizados para *forms.* Estos también deben agregarse como atributos al elemento Html objetivo. El código a ejecutar al dispararse cada uno de estos eventos es código Links, entre llaves, de igual manera que los atributos l:xx de los *forms*.

Propiedades de los eventos de usuario

Al igual que en Html, es posible extraer información de la acción del usuario. Esta información viene en la forma de la variable *event* en Links. Sus propiedades son:

- *getTarget*
- *getTargetElement*
- *getFromElement*
- *getToElement*
- *getPageX*
- *getPageY*

Por ejemplo *getFromElement* y *getToElement* devuelven los elementos desde y hacia donde ocurrió el evento *mouseout*.

3.7 Modificando la página web

Links permite actuar como un servidor web, sirviendo páginas originalmente creadas en su tipo básico Xml. Así un valor Xml, puede ser usado para crear una página completa. Pero no solo esto se puede hacer, también es posible reemplazar una parte de la página web actual, por Xml.
Links distingue entre el DOM [37] de la página web actual, y el tipo Xml. El DOM representa a la página web, que está siendo desplegada por el navegador, esta puede cambiar incluso por el propio navegador. El tipo Xml es un tipo de Links inmutable, que permite ser traducido a una página web. Links provee funciones para modificar el DOM insertando valores Xml. Nuevamente estas funciones del DOM, deben ser entendidas por el navegador, que solo entiende Html-Javascript, por lo tanto, deberían ser muy similares a las funciones Javascript del manejo del DOM.

Funciones sobre el DOM

Estas son algunas de las funciones de manejo del DOM:

- appendChildren(xml, parentNode)
- replaceNode(xml, node)
- replaceDocument(xmlVal)
- getNodeById(id)

Por un listado más completo puede consultarse [1]

Funciones sobre Xml

Estas son las funciones provistas para manejo de Xml:

- getTagName(xml)
- getTagName(<h1>Dog Bites Man</h1>)
- getTextContent(xml)
- getAttributes(xml)
- getAttributes( )

- hasAttribute(xml,attrName)
- hasAttribute(, "class")
- getAttribute(xml,attrName)
- getAttribute(, "class")
- getChildNodes(xml)

Cookies y almacenamiento en el cliente

Existen dos operaciones para manejar cookies:
getCookie : (String) ~> String
setCookie : (String, String) ~> unit

La primera permite recuperar un cookie por su nombre, y la segunda le asigna un valor. No se puede indicar la fecha de expiración, y al cerrar la ventana del navegador se borra el cookie.
Links no tiene soporte para *sessionStorage* ni *localStorage*.

Javascript nativo

Destacamos que recientemente es posible embeber Javascript nativo dentro de Xml. Esta característica va en contra del espíritu de Links, y debería usarse solo como último recurso.

3.8 Funciones incorporadas

Links posee una variedad de funciones integradas, algunas ya mencionadas antes, como *implode* y *explode*, que permiten convertir desde *String* a *[Char]* y viceversa. También existen otras funciones de conversión entre los distintos tipos. Algunas otras funciones ya aparecieron en los segmentos de código anteriores, como *print*, que muestra un String en la salida estándar, también existen otras para reportes de errores como *error* y *debug*.
Un listado con estas funciones se puede consultar ejecutando:
links> @builtins;

desde el interactivo.
Algunas funciones merecen una mención especial, como:

- *freshResource : () ~> ()*
  En el contexto de una página web, en caso de recargarse la página, con F5 por ejemplo, la ejecución de la misma ocurre desde *freshResource* en adelante. Esto es útil cuando la misma tiene acciones que no deberían volver a ejecutarse, como modificaciones a la base de datos. Esto es consecuencia de que el estado está almacenado en la URI, como se puede ver en la sección continuaciones, de donde una actualización de la página vuelve a ejecutar las mismas acciones que llevaron a ella.
- *sendSuspend : (((a) -> Page) d~> Page) d~> a*
  El uso de continuaciones incluido en *sendSuspend*, permite simplificar en gran medida la incorporación del login a una aplicación, por ejemplo. Esta función suspende la ejecución actual, y navega hacia otra página, desde la cual se puede volver a la ejecución suspendida, volviendo a ejecutarse desde el punto de la

llamada, sin necesidad de haber almacenado ninguna información extra. Más detalles se pueden ver en la sección continuaciones.

3.9 Módulos

Links incorpora un sistema de módulos [30] desde fines del 2016 [31]. Si se quiere llamar a una función del archivo *myModule.links*, de nombre *myFunModule()*:

```
open myModule
myModule.myFunModule()
```

Para poder utilizar el sistema de módulos, se debe llamar al programa que los usa con el parámetro -m:
linx -m myProgram.links

3.10 Configurando el servidor web

Links incorpora un servidor web [30], que permite mostrar las páginas (*Page*) generadas por las funciones de los programas. Un mismo servidor web, puede mostrar páginas generadas por distintos programas, gracias a la funcionalidad de módulos, explicada anteriormente. Para permitir mostrar distintas páginas, es necesario incluir un mapa, que haga corresponder a cada página web generada que se quiera mostrar, una dirección web distinta. Esto se hace con la función *addRoute*, que mapea un directorio de la URI, a la función.
Por ejemplo si en:
http://localhost:8080/myDir

se quiere mostrar la página web generada por la función

```
fun myFun() {
  page
    <html>
     ...
    </html>
}
```

Se debe llamar addRoute con:
addRoute("myDir", fun(_) {myFun()})

También es posible incluir contenido estático, que no está escrito en Links, como bibliotecas Javascript, archivos Css, u otro contenido. Esto se realiza mediante:
addStaticRoute(uri, path, mimeTypes)

donde se hace corresponder las direcciones que empiezan con *uri*, con las direcciones que empiezan con *path*, indicando además una lista de pares (*extensiones, myType)*, para que el navegador muestre eventualmente el archivo adecuadamente.

El puerto por defecto, 8080, también puede ser cambiado, agregando al archivo de configuración (ver Anexo A) la palabra *port*:
port=80

Una vez llamadas las anteriores funciones las veces que sea necesario para configurar el servidor, se deben servir las páginas web con:
servePages()

3.11 Corriendo Links

Los programas escritos en Links se pueden ejecutar directamente en Links, o en un ambiente interactivo similar al de Haskell.
Para correr directamente un programa escrito en Links, se debe ejecutar:
linx nombreArchivo

El archivo contendrá un conjunto de definiciones en Links, y además deberá realizar el llamado a una función. Esta función será el punto de partida en la ejecución del programa. No solo es posible ejecutar aplicaciones web, también es posible ejecutar programas sencillos, del lado del servidor.

Si el archivo *myProgram.links* contiene:

```
fun myFun(x) {
  x+2
}
myFun(1)
```

y se ejecuta en una terminal del sistema:

```
linx myProgram.links
```

se mostrará en la salida estándar:
3

y el programa terminará.

Si en cambio se quiere correr el servidor web, y mostrar una página, se debe configurar previamente como se indicó en el apartado anterior. Así por ejemplo:

```
# página principal de la aplicación
fun mainPage() {
  var myPage= page  <#></#>;
  layout(myPage)
}

fun main()
{
```

```
  addStaticRoute("/css/", "css/", []); # cargamos los css del proyecto
  addRoute("", fun(_) {mainPage()});
  servePages()
}

main()
```

configura el servidor, y luego sirve las páginas desde la raíz, además de proveer archivos en forma estática, desde el directorio *css* y la dirección *css*.

Para correr el interactivo, basta ejecutar el comando *linx* sin parámetros.
Desde el interactivo, es posible obtener cierta información del ambiente, como funciones incorporadas al lenguaje, directivas, etc [30]. Por ejemplo con:
links> @settings;

es posible obtener información de configuración del ambiente, o con @builtins, es posible tener un listado de las funciones nativas del lenguaje, como se indicó previamente.

3.12 Continuaciones y programación distribuida

Links utiliza una variación de *continuation-passing style* en la compilación de sus funciones, que permite el uso de continuaciones [2]. Las continuaciones en Links están inspiradas en PLT Scheme [2], actual Racket [32].

Los programas en Links pueden verse como programas distribuidos, que pueden ser ejecutados en el cliente o en el servidor. Links permite ejecutar transparentemente para el programador funciones de estos programas, o sea desde funciones en el servidor se pueden ejecutar funciones en el cliente, y desde el cliente invocar a funciones en el servidor. El programador no ve diferencia en el modo de invocarlas [6].
Las continuaciones son, dada la actual ejecución de un programa en un punto, poder almacenar todo el estado de ejecución en ese punto, de manera de poder volver a ejecutar el programa desde exactamente ese punto sin que nada más hubiera pasado [3, 4].
Esto es especialmente útil para el caso de la llamadas entre el cliente y el servidor.
Si *f* es una función en el servidor, *g* es una función en el cliente, *f* hace internamente una llamada a *g* y se llama desde el cliente a *f*, ocurre lo que se diagrama en el Figura 1, extraída de [2]. Cuando el cliente llama a *f* en el servidor, ésta a su vez llega al punto donde se llama a *g*. Allí, se debe volver al cliente, pero para llamar a la función *g*, no al punto siguiente donde se llamó a *f*. Para esto, se responde al cliente con una continuación del servidor, en el punto de invocación a *g*. Esta continuación es *k*, y contiene todo la información de contexto del servidor, permitiendo volver a invocarlo desde ese punto. Al responder al cliente, nada es almacenado en el servidor. Cuando el cliente recibe la respuesta, con la continuación, allí se indica además que debe llamar a *g*, y volver a llamar al servidor, con la continuación y el valor resultado de *g*, (*r*). Así al volver a llamar al servidor con estos valores el mismo vuelve a ejecutar el programa exactamente en el punto donde había sido llamada *g*, ahora con el valor resultado. Una vez finalizada la ejecución de *f*, su resultado vuelve al cliente. Esto permite transparentar el uso de funciones desde el servidor

y cliente, hacia el cliente y servidor. No hay diferencia para el programador entre una llamada a una función en el cliente y una llamada a una función en el servidor, es transparente para él.

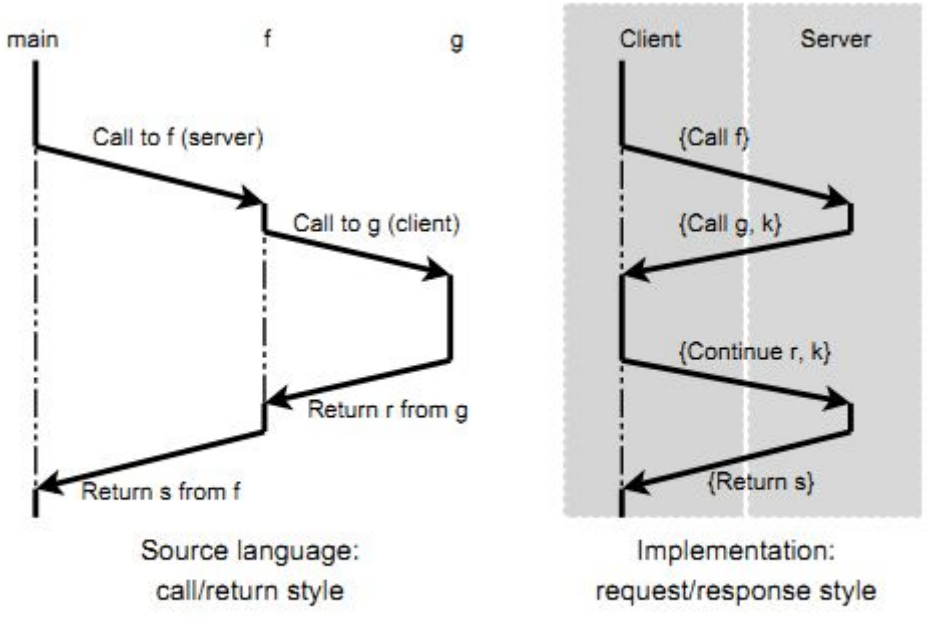

Figura 1 [2]

Las ventajas de legibilidad, mantenibilidad, son muy grandes, no es necesario pensar en qué protocolo usar, cuando, que funciones llamar, todo esto es transparente para el programador, hay un alto nivel de abstracción de elementos que no son centrales al desarrollo.
Tambień hay situaciones, como el login, donde es especialmente útil el uso de continuaciones. En una aplicación web con autenticación, toda página debe verificar que el usuario esté autenticado, para poder acceder a ella. Si no está autenticado, se debe navegar a la página de login, y luego lo ideal sería volver a la página que originalmente había requerido el usuario. Esto se puede realizar con continuaciones. Dada una página cualquiera de la aplicación, se puede almacenar su continuación, justo al comienzo de la misma. Luego invocarse una función, que verifique que el usuario está autenticado, y mientras no lo esté pedir autenticación. Cuando finalmente esté autenticado, esta función puede usar la continuación, para volver al punto original donde estaba la página que pidió la verificación de la autenticación, sin que nada más haya pasado. La página de login, no "navegó" hacia la página original, no hubo una navegación explícita, tradicional, sino que se volvió al mismo punto de ejecución del programa, en que estaba originalmente. Se alteró el flujo lineal común de ejecución del programa. Esto es lo que hace *sendSuspend*, es posible llamar a una página web, a la vez de enviarle la continuación actual, y que la página decida cuándo aplicar esa continuación y volver a la ejecución normal del programa.
Como se puede ver, las continuaciones son un mecanismo muy poderoso. Se usan para implementar concurrencia y *formlets* [18].

En cuanto al manejo de la continuación, en distintos lenguajes se puede almacenar en el servidor, o en el cliente. Links serializa la continuación en un string, y la almacena enteramente en el cliente. Cada vez que hay una comunicación con el servidor, se envía la continuación [18].

3.13 Efectos y Manejadores

Los efectos en Links originalmente sólo disponían de los tipos *wild* para compilación de código Links a SQL y tipo *hear:A* asociado a la concurrencia [9]. Más recientemente (fines 2017 [31]) se incorporaron características más avanzadas que permiten que el programador defina efectos personalizados. Estos efectos tienen características muy poderosas, ya que permiten el manejo de errores como una alternativa a las variantes del propio Links, y tienen su equivalente en algunas mónadas en Haskell [16], y las excepciones en Java [9]. También permiten explorar la ejecución de un programa evaluando distintas computaciones. Asociados a estos efectos hay operadores y manejadores. Los operadores identifican el efecto y los manejadores incorporan alternativas de ejecución del operador. Veamos algunos ejemplos.

Un primer ejemplo es un manejador simple de error, una excepción. Si la operación dió error, devolverá Nothing, y si es correcta devolverá Just x.

```
sig myDiv : () {Fail|_}-> Int
fun myDiv() {
  var x = 1;
  var y = 0;
  if (y == 0) {
    do Fail;
    0
  } else {
    x/y
  }
}

# manejador de la excepción Fail, en cuyo caso devuelve Nothing
handler maybeResult {
  case Return(x) -> Just(x)
  case Fail(_) -> Nothing
}

links> maybeResult(myDiv)() == Nothing;
true : Bool
```

La función *myDiv* realiza divisiones de enteros. Si *y es* 0, devolverá un error. Con la primitiva *do* se llama al operador Fail, que genera un efecto del mismo nombre, Fail, como se puede ver en su firma. El manejador *maybeResult,* recibe como parámetro la función *myDiv,* y son ejecutadas las opciones *case*, en caso de que se llame al operador *Fail*, como se hace en

do Fail, o en caso que la función finalice y devuelva un valor. Esto se realiza por *pattern matching*, el operador *Fail* se corresponde a la opción *case Fail(_)* y la finalización de la función a *Return(x)*, el valor de x es el valor devuelto por la función *myDiv*.
Así de cierta manera *maybeResult* 'envuelve' a la función *myDiv*, y devuelve un tipo personalizado de acuerdo a los deseos del programador. La alternativa similar al uso del tipo *Maybe* en Haskell, es que la función *myDiv* devolviera el tipo *[|Nothing | Just:Int|]*, en cuyo caso este tipo debe ser manejado directamente por la función, en vez de realizarse por un manejador de la excepción.

Los manejadores de efectos también permiten continuaciones y parámetros [21]:

```
sig moneda : () {Choose:Bool|_}-> [|Cara|Cruz|_|]
fun moneda () {
  if (do Choose)
    Cara
  else
    Cruz
}

handler randomResult {
  case Return (x) -> x
  case Choose (resume) -> resume (random() > 0.5)
}

links> randomResult(moneda)();
Cruz : [|Cara|Cruz|_|]
```

En este caso, a pesar que el operador *Choose* no recibe explícitamente un parámetro, se envía como parámetro la continuación en ese punto del programa. Por lo que es posible para el manejador devolver un valor de ese operador. Así en este ejemplo puede verse que al llegar al operador *Choose* el manejador genera un número aleatorio y devuelve *true* o *false* en función de ese número. Entonces la función moneda devuelve *Cara* o *Cruz*, y el manejador en el case *Return(x)* devuelve el propio valor.

De esta manera, la misma función moneda, puede correrse con distintos manejadores, que permiten evaluarla en distintas formas.
Incluso más en general se pueden crear estrategias de computación de funciones más elaboradas [16].

3.14 Tipos Sesión

Tipos de sesión es una característica muy avanzada del lenguaje que ha sido incorporada recientemente [31], y que permite abstraer a un nivel muy importante conceptos como el de protocolo de comunicaciones. Mediante la definición de un tipo es posible establecer qué forma debe tener la comunicación entre un cliente y un servidor que está brindando ciertos servicios específicos al cliente. También involucra una alta complejidad y es aún fuente de

importante investigación [33] por lo que brindaremos solo una idea conceptual general para mostrar la capacidad del lenguaje.

Los tipos de sesión son tipos para protocolos de comunicación. Permiten garantizar de forma estática que la comunicación cumple con cierto protocolo. Especifican el tipo de los datos y el orden en la comunicación. Veamos un ejemplo [26]:

Se quiere realizar una autenticación en dos fases, si el intento de autenticación es desde un dispositivo no reconocido, entonces se le envía un código de verificación (*challenge*) luego se responde con este código de verificación procesado mediante una llave física.
El tipo de esta comunicación sería:

```
TwoFactorServer ≜
  ?(Username, Password).⊕{
    Authenticated : Main,
    TwoFactorChallenge : !Challenge.?Response.
      ⊕{Authenticated : Main, AccessDenied : End},
    AccessDenied : End}
```

Este tipo especifica que primero se debe recibir (*?*) un par *username* y *password* desde el cliente. A continuación se elige entre autenticación exitosa (*Main*), realizar la verificación en dos pasos (*TwoFactorChallenge*) y denegar el acceso (*AccessDenied*). Si se debe realizar la autenticación en dos pasos, se le envía al cliente el código de verificación (*!*) se espera la respuesta y se autentica o deniega el acceso definitivamente.

Hay muchos detalles involucrados con esta característica del lenguaje como los tipos de canales de comunicación, todos involucran una importante complejidad. La implementación de Links, también permite el manejo de errores en la comunicación, por primera vez en un lenguaje funcional [26]:
try L as x in M otherwise N

donde *L* puede contener una función de sesión, y *M* y *N* son *endPoints*.

Los tipos de sesión son especialmente útiles cuando el protocolo está definido, como en un chat server.

4 Caso de Estudio

4.1 Especificación del caso de estudio

Para poder evaluar la efectividad del lenguaje Links, se plantea la creación de una aplicación web típica. Se tratará de alcanzar los aspectos comunes de una aplicación web, como altas, bajas y modificaciones sobre una entidad, persistencia en una base de datos, validación, reportes, acceso a servicios web, autenticación, Ajax. Más adelante se agregarán más detalles sobre la justificación del producto. Veamos las especificaciones.

Se trata de una aplicación web donde administrador y estudiantes se pueden autenticar en una página. Pueden cerrar la sesión en cualquier momento, en cuyo caso volverán a la página de autenticación. Los estudiantes una vez que ingresan al sitio, pueden inscribirse a los cursos del mismo. El administrador, y solo él, podrá realizar operaciones de alta, baja y modificación sobre los estudiantes y sobre los cursos.
Cuando el administrador ingresa un estudiante en el sistema, los datos a ingresar serán:

a. usuario (único en el sistema, obligatorio)
b. contraseña (obligatorio)
c. confirmación contraseña (obligatorio)
d. nombre (obligatorio)
e. apellido (obligatorio)
f. fecha nacimiento (obligatorio)
g. cédula identidad (única en el sistema, obligatorio)
h. sexo (obligatorio)
i. nacionalidad (Uruguayo, Argentino, Brasileño. Opcional.)

Sólo disponible para el administrador, existirá la opción de listado de estudiantes, donde se listarán ordenados por CI. Desde esta lista se podrán modificar y borrar los estudiantes. Al modificar un estudiante, todos sus datos se podrán cambiar. Al borrar uno, se pedirá confirmación y se recargará el listado.

Al ingresar un curso al sistema, este constará únicamente del campo nombre. También existirá un listado de cursos donde análogamente a estudiantes se podrá modificar y borrar. No se podrá borrar un curso que tenga estudiantes inscriptos.

Al ingresar un estudiante al sistema aparecerá la opción de listado de todos los cursos. Se podrá inscribir y desinscribir a los mismos. Se resaltarán los cursos en los que está inscripto en rojo. Luego de inscribirse/desinscribirse, se mantendrá en la misma página de listado de cursos. Esta página no se recargará toda, sino solamente la parte de listado de cursos.
Habrá una sección reportes solo disponible para el administrador, donde existirá un reporte que mostrará por estudiante, la cantidad de materias que está cursando.
Además en todo momento, desde un servicio web, se mostrará la temperatura en Montevideo.

También la aplicación deberá correr en tanto computadoras de escritorio como celulares, mostrándose una interfaz aceptable al usuario.

Justificación de elección de los requerimientos

Como recién se mencionó, la idea es crear una aplicación web típica que explore los aspectos más comunes del desarrollo web. Así el login es una de ellas, es un requerimiento muy común y debiera estar presente en la aplicación.
En cuanto a la creación de los usuarios, se siguió la idea de varios sistemas donde se asigna una clave temporal, y luego el usuario debe cambiarla la primera vez que ingresa al sistema. Como la parte de cambio de la contraseña solo agrega complejidad al producto y no aporta en cuanto a la exploración de las funcionalidades disponibles, no se incluyó en las especificaciones.
Con respecto a las relaciones entre las entidades, como se ve en la sección siguiente, se establecen relaciones uno a muchos y muchos a muchos, lo que debería representar significativamente una aplicación típica.
En cuanto a los tipos de datos de las entidades, se trató de incluir los más usados. También se manejaron las distintas opciones en cuanto a cardinalidad de los campos de las entidades. Así algunos son obligatorios, y otros opcionales. La idea con el campo *sexo* era que fuera un enumerado no opcional, así su representación gráfica debería ser un conjunto de *radio buttons*. En este sentido también surgen los otros campos, para explorar todos los componentes gráficos posibles en una Html *form*. Los enumerados opcionales se representan con el componente Html *select*. Mención especial merece el campo tipo *date*, con el cual deberían explorarse sus posibles representaciones. Situaciones similares ocurren con el agregar una contraseña y su verificación.
Con relación a la validación, se trató de incluir todas las validaciones típicas, como campos requeridos, largo de campos, duplicado de claves, validación de enteros, validación de fecha, etc.
La posibilidad de inscripción junto con no refrescar la página enteramente, introduce el uso de Ajax, más manejo del DOM de una página web.
El mostrar la temperatura, evalúa la posibilidad de uso de servicios web externos.
También se agregó el requerimiento no funcional de que tenga una interfaz aceptable con el usuario en dispositivos de escritorio y móviles, o sea la aplicación debe ser *responsive.* Actualmente los dispositivos móviles constituyen una parte significativa del mercado, por lo que pareció adecuado este requerimiento.

4.2 Implementación

Para el desarrollo del producto especificado, fue necesaria una primera etapa de aprendizaje del lenguaje. Esto incluyó la lectura del material disponible sobre el lenguaje, junto a la realización de casos de prueba sobre el mismo. La intención de la realización de los casos de prueba llevaron a la necesidad de crear un ambiente de desarrollo, más la investigación de las funciones necesarias para poder crear este ambiente de pruebas en el lenguaje. Dada la carencia en Links de funciones propias de asistencia a las pruebas, se tuvieron que crear funciones de ayuda para pruebas unitarias, como *myAssert*.

Esta primera etapa involucró la exploración de todos los aspectos de Links y por lo tanto gran parte de sus funciones. Este trabajo está distribuido en tres archivos:

- *pruebas.links* contiene las primeras pruebas unitarias sobre el lenguaje, para aprendizaje y evaluación del mismo.
- *web.links* contiene pruebas web.
- *pruebasEfects.links* contiene pruebas sobre el sistema de efectos en Links.
- *testVarSignatureWebFailure.links* muestra fallas en el sistema de módulos.

Las pruebas web pueden accederse a través de:
http://abentan.ddns.net:8080/pruebas

Los otros archivos fuente pueden encontrarse en la documentación del proyecto.

En cuanto al ambiente de desarrollo tuvo que instalarse un sistema con Linux y además un servidor de producción con el mismo sistema. Se usó como SCM Git y respaldo en la nube con Bitbucket. Más detalles pueden verse en el Anexo B.

Links sólo dispone oficialmente de Postgresql como motor de base de datos por lo tanto se utilizó este sistema.

Links tampoco dispone de IDE, por lo que tuvo que investigarse editores adecuados. Más información puede encontrarse en el Anexo B.

El caso de estudio, producto principal, puede encontrarse en:
http://abentan.ddns.net:8080/

Para correr las pruebas unitarias, debe correrse sin la opción de configuración: shredding=on, y para correrse el caso de estudio debe activarse esta opción. Esto es debido a errores propios internos del lenguaje, que llevan a que no compile el programa de pruebas si se activa la opción. Más detalles puede verse en la sección 5.18, en la discusión sobre el acceso a la base de datos de Links.

La implementación del caso de estudio se vio simplificada por la realización de la primera fase del proyecto, que involucró aprendizaje y estas pruebas unitarias. Un esquema de la base de datos de este caso de estudio se encuentra a continuación:

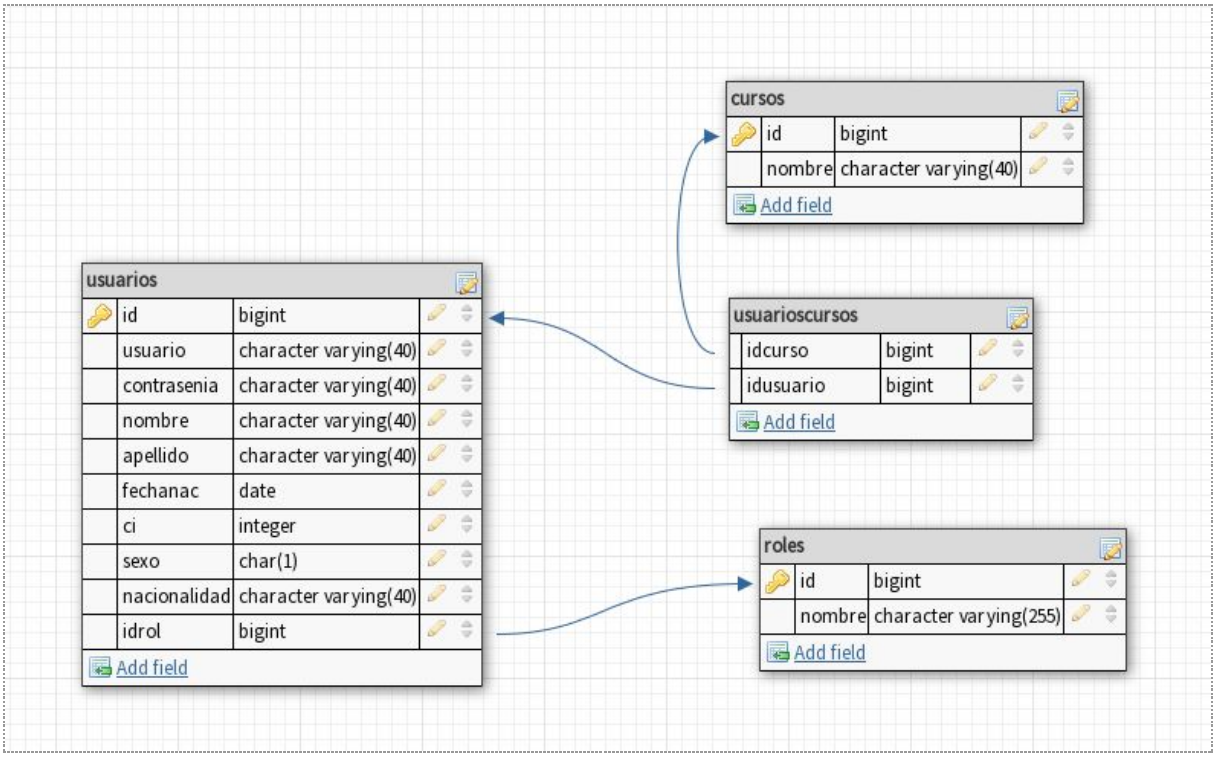

Figura 2. Diseño de la base de datos.

El login se implementó usando *sendSuspend* previamente explicado en la introducción al lenguaje.

Para cumplir con el requerimiento *responsive* se incluyó uno de los frameworks Html-Css más comunes, Bootstrap 3 [39]. Aquí puede verse un ejemplo con distintos tamaños de pantalla:

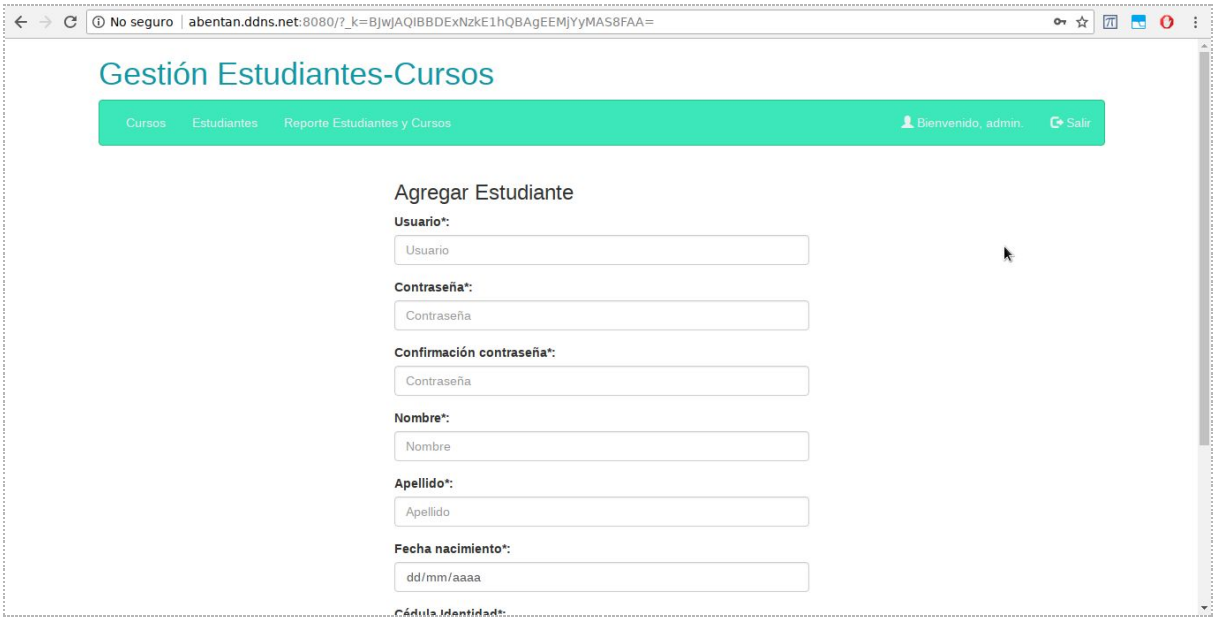

Figura 3. Agregar estudiante en PC escritorio.

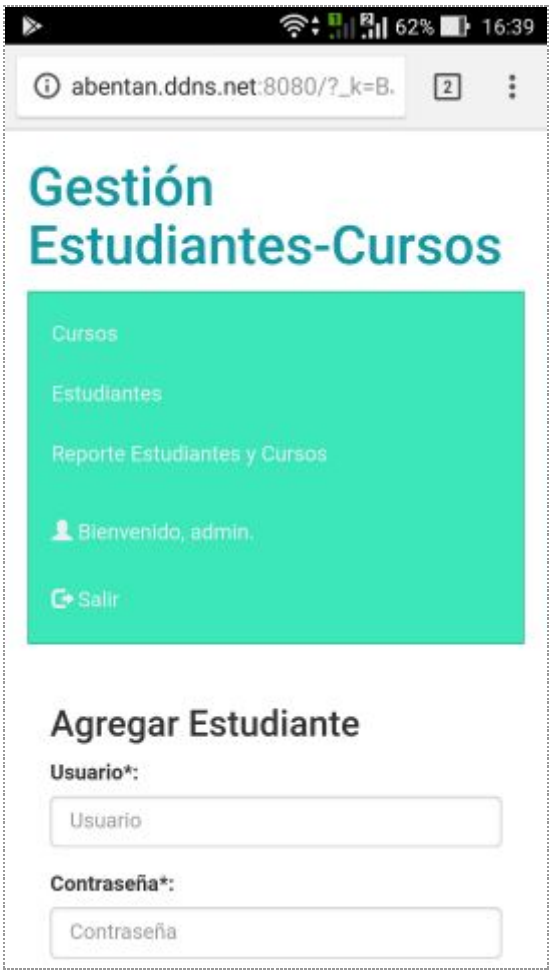

Figura 4. Agregar estudiante en móvil.

Como se puede ver se agregó una barra de menú, que cambia según el tipo de usuario que ingrese al sistema. Esta barra de menú se despliega adecuadamente en dispositivos de escritorio, pero no se contrae como sería esperable en móviles.
En cuanto a los campos en los *forms*, para el *layout* se usaron los Html con Css de Bootstrap. Fue posible realizar el *layout* con este framework. En general en cuanto a estos campos, no se eligió usar formlets, como se explicará más adelante en la sección correspondiente.
Con respecto al campo *fecha de nacimiento*, originalmente se pensó en usar tres campos con enteros, como se sugiere en la sección introducción al lenguaje. Finalmente se optó por un *input* con *type="date"*, y que el navegador muestre el componente adecuado, disponible desde Html 5 [40], más un tipo *String* por el lado del lenguaje, y un tipo *Date* en la base de datos. Dado que Links no posee tipo *Date*. Html 5 asegura compatibilidad entre el formato *date-string* y el mismo en la mayoría de las bases de datos [40]. Más adelante se puede encontrar más información.
Con respecto al campo *sexo*, para el cual se pretendían usar *radio buttons*, no fue posible usarlos, como se explicará en la sección Html. Se utilizó en su lugar un *select*.
En el reporte de estudiantes-cursos, no pudo realizarse un única consulta para contar los cursos por estudiante, sino que se creó una consulta anidada que para cada estudiante trae su lista de cursos. Luego con una función map se cuentan los cursos.

Algo similar ocurrió en la página de inscripción a cursos del estudiante. En este caso se realizó otra consulta anidada trayendo todas las inscripciones del estudiante actualmente autenticado, para cada curso. Luego se usó map para cambiar la lista a una lista de cursos y valor inscripto (verdadero o falso) como se desprende del modelo de la página.

En lugar de usar Ajax, se utilizó submit para recargar solo un elemento de la página. Se verán detalles en el capítulo discusión.

No pudo agregarse información de la temperatura dado que Links no dispone de comunicación con servicios web.

Cabe notar que la suma de los archivos de pruebas unitarias del lenguaje, más los archivos de pruebas web, más el archivo de caso de estudio (900 líneas), suman 1800 líneas de código, sin incluir la parte de Css. Teniendo en cuenta que el proyecto de más volumen reportado son 4500 líneas [9], estamos ante un proyecto de tamaño significativo.

5 Discusión

En este capítulo se tratará de evaluar el lenguaje, empezaremos por analizar los aspectos más generales del mismo, para ir profundizando cada vez en más detalles, relevantes al desarrollo web. Este análisis intentará extraer conclusiones clave en cuanto a su efectividad, como ser aspectos de la calidad del producto y su desarrollo: legibilidad, mantenibilidad, tendencia a la correctitud, factibilidad de requerimientos, performance, seguridad, tiempo de desarrollo, etc. Buscaremos recorrer la mayor cantidad posible estos aspectos, intentando tener una visión lo más abarcativa posible, a partir de todo el estudio previo: estudio del lenguaje, caso de estudio y bibliografía relacionada. Trataremos de finalmente llegar a conclusiones que integren los distintos aspectos analizados.

5.1 *Impedance mismatch problem*

Quizás el aspecto más importante del lenguaje, es su intención de resolver este problema, de la 'falta de concordancia' entre los diferentes lenguajes usados en la programación web actual, a través de usar solo Links. Esta característica lleva a ver aspectos como traducción de Links en el front-end, a Javascript, Html y Css, y su conversión en el back-end, a SQL.

Analizemos primero las grandes y profundas ventajas que en general tiene esta característica.

Solo es necesario aprender un lenguaje. En la programación web tradicional, es necesario aprender por lo menos 3 lenguajes, con sus respectivas sintaxis, palabras clave, sistema de tipos, etc. Esto lleva a que incluso haya especializaciones en este aspecto, programadores que son especialistas en Javascript por ejemplo. Nada de esto es necesario en Links, solo se aprende un lenguaje, de sintaxis relativamente sencilla, y en ese sentido no muy diferente a otros.

La conversión de tipos en principio tampoco es necesaria, todos los tipos están dentro de Links, por lo tanto no sería necesario convertirlos. Esto también es como idea una gran ventaja, no es necesario crear ningún convertidor de tipos, como tanto es necesario en la programación web tradicional, por ejemplo Java.

También aparecen otras ventajas, al haber únicamente un lenguaje, y las consultas a la base de datos estar escritas en el mismo, estas consultas son sujeto de chequeo de tipos y de sintaxis, en lugar de ser un string embebido, y lo mismo ocurre con Javascript. En cuanto a estos puntos, entraremos en más detalles en las secciones siguientes.

A primera vista, estas ventajas son de gran peso en la evaluación del lenguaje con respecto al enfoque tradicional:

- legibilidad, la mejoran, no es necesario aprender varios lenguajes, todo es más sencillo de entender, no hay llamadas remotas explícitas, etc

- mantenibilidad, al lograr este mayor nivel de abstracción, los programas son mucho más cortos, y también se nota una relación más directa y clara entre las funciones del cliente y servidor, junto con la legibilidad, se mejora mucho la mantenibilidad
- tiempo de desarrollo, también lo mejora, un solo lenguaje para aprender

Estas ventajas, solo derivadas de este punto, son de gran peso. Estas características son muy importantes en cuanto a la evaluación global del lenguaje.

Estas ventajas, que a primera vista son trascendentes, quedan supeditadas a en qué medida se logra el objetivo de usar un solo lenguaje, y no 3. ¿Realmente se puede escribir cualquier código Html en Links?, ¿y Css?, ¿y Javascript?, y ¿Sql?. Y el código generado, ¿tiene la misma performance que en programación tradicional?. ¿Se puede resolver cualquier requerimiento en Links? ¿Realmente es escalable? Aparecen preguntas relativas a la expresividad del lenguaje principalmente, que hablan de su factibilidad ante distintos requerimientos, y de performance. Esto es lo que veremos en las secciones siguientes, junto con otras características más allá del *impedance mismatch problem*.

5.2 Tipado fuerte

Esta es otra característica de Links, de gran peso en su evaluación. Son muy conocidos los inconvenientes del tipado débil en el front-end: el navegador prácticamente acepta todo, y en todo caso mostrará una página web defectuosa, pero la muestra. Se puede escribir Html sin cerrar tags, con tags que no existen, y hasta no verlo en el navegador no se sabrá si lo que se hizo estaba bien o mal. Lo mismo ocurre con Css, que puede contener valores no propios del lenguaje, y Javascript, donde hasta no correrlo no se sabe si está bien, y además, si incluso corre se tienen otros problemas como el tipado débil del mismo lenguaje. Lo mismo ocurre con el back-end por el lado de Sql. En la mayoría de los casos, se usa Sql como un String, sin ningún chequeo de tipos y sintaxis. Todo este gran problema, quizás el más importante viene desde Javascript, lleva a que muchos errores recién se conozcan en tiempo de ejecución, e incluso en ese caso todavía falten muchos por detectar.

Nada de esto ocurre en Links gracias a su tipado fuerte. Todos los errores de tipos y de sintaxis se conocen en tiempo de compilación, lo que ahorra mucho tiempo de desarrollo en general y pruebas en particular. El peso de esta ventaja es muy grande, y es una característica que en principio tiene muy pocas desventajas: es necesario convertir los valores de un tipo a otro, en caso de querer usarse en ciertas funciones, algo menor comparado con sus ventajas. Esto aporta fuertemente a la correctitud del producto, también mejora legibilidad, y mantenibilidad. Esta característica afecta a muchas otras que se van a analizar, como las que mencionamos previamente.

5.3 Programación distribuida

Esta es otra característica del lenguaje que afecta toda la programación en el mismo. Cualquier programa se beneficia por la transparencia de programación en cuanto a su localización.

Típicamente en Javascript, es necesario crear un objeto Json, luego utilizar Ajax para enviarlo al servidor, y además implementar un *callback* para manejar la respuesta. Todo esto es abstraído por Links, no es necesario preocuparse por ninguno de estos elementos. Simplemente se hace la llamada como a una función más, no es necesario indicar si es en el servidor o en el cliente.
Esta facilidad para la implementación, también puede tener características negativas. Por ejemplo si una función realiza llamadas a la base de datos, luego modifica la página, luego otra llamada a la base de datos y luego modifica la página nuevamente, estamos hablando de varios llamados al servidor, lo que podría ser ineficiente. Si el programador es consciente de estos llamados, podría tratar de implementar la función primero con los llamados a la base de datos, y que se hagan en un único ajax, y luego realizar los cambios en la página.

Esta característica del lenguaje brinda grandes ventajas: mejora ampliamente la legibilidad, mantenibilidad y tiempo de desarrollo. Por otro lado, puede afectar la performance, dependiendo del estilo del desarrollador. También, al estar involucrada en la comunicación, cualquier error que se produzca en el mismo, podría tener como causa parcial, por lo menos, esta característica del lenguaje. Así, los problemas con *replaceNode* después de una llamada a la base de datos, que se verán en la siguiente sección, podrían tener que ver con problemas de implementación de la programación distribuida. Esto no lo podemos responder con el alcance de este trabajo.

5.4 Html - Css - Ajax

Una vez que hablamos de lo más general, Html parece el primer punto para seguir. Una aplicación web, debe tener una página web. Debe generar Html, con su lenguaje de estilos fuertemente acoplado a él, Css. En Links se escribe código que genera Html y Css, analizaremos este punto.

Ventajas

Hay varias e importantes ventajas en la generación de Html en Links. Links genera Html a través de Xml. Xml es a su vez un tipo nativo en Links, se puede escribir Xml directamente en Links, convertir distintos tipos a y desde Xml, y manejarlo como un valor más. Esto tiene las siguientes ventajas.

Html

Permite generar rápidamente Html. No es necesario escribir un *string* o leerlo desde un archivo, y luego convertirlo a Xml, como en lenguajes que no lo tienen embebido, como Java. Esto acelera el desarrollo.

Está embebido el tipo Xml en Links, por lo tanto muchos errores de Html no ocurren. Siempre se deben cerrar todas las *tags* y en el orden correcto. Esto es consecuencia del chequeo de tipos.

El código Links puede estar embebido en Xml. A su vez, Xml puede contener código Links embebido, que genere otro Xml. Esto brinda en principio posibilidades de *templates*. Se

puede diseñar una página web, donde solamente se cambie cierto segmento de código, y no toda la página cada vez. Esto se implementó en el caso de estudio con una función *layout*, que recibe el segmento de página central, y luego devuelve la página completa, agregando siempre el mismo *head*, con todas las bibliotecas Javascript y Css, y con el mismo menú. Implementar esto tradicionalmente es mucho más complejo. En Struts [41] se usa típicamente Tiles [42], lo que involucra el uso de más bibliotecas Java, una sintaxis particular que hay que aprender, y configuración. Otra solución más moderna con Node.js [43], es Mustache.js [44]. Esta opción también involucra una sintaxis propia, con generación de código en Javascript, lo que tiene más inconvenientes. Otras opciones se pueden implementar directamente con Node.js y otras bibliotecas, como sustitución de código, pero tienen similares desventajas. Esto es una muy fuerte ventaja, que acelera el desarrollo y permite encapsular segmentos de código fácilmente. Esta característica tiene ciertas limitaciones que se analizarán más adelante.

Al ser las funciones valores de primera clase, es posible usarlas fácilmente en validaciones en páginas web. Por ejemplo al inscribirse un estudiante en un curso, se debe hacer la misma validación por si el curso existe, y si existe, realizar la operación de inscripción o desistir. Se pasa esta operación como una función directamente al validador, que la ejecuta sin necesidad de otros elementos extra.

Css

Algo similar ocurre con Css. En Xml, mediante la *tag* de Html <*style*> en el elemento <*head*>, se permite agregar Css internos, código Css en la propia página web. También existen similares ventajas.

Se permite generar rápidamente Css. Al estar Css embebido en Xml, se puede generar de igual manera.

Se puede embeber código Links dentro de Css. Son conocidas las limitaciones de Css, no es posible establecer herencia de clases, la capacidad de uso de variables es escasa, no es posible crear funciones, ni hacer cálculos, etc [47]. En estos casos una posibilidad popular es usar Less.js [45] o Sass [46]. Esto trae aparejado un procesamiento de los archivos Less o Sass para luego convertirlos a Css, en un ambiente como Node.js. Nuevamente involucran el uso de más herramientas, externas al lenguaje original, lo que se traduce en más aprendizaje de sintaxis, tipos, en definitiva otro lenguaje. Esto no es necesario en Links, el propio lenguaje, permite encapsular segmentos de código Css, así creando herencias o funciones. Por ejemplo se puede crear una función que devuelva un color:

```
.bb {{ {
  color()
} }}
fun color () {
  stringToXml("color:black;")
}
```

Esto se puede ver en web.links, sobre la función *mainPage*.

Limitantes

Junto a estas ventajas, también existen desventajas. Muchas surgen de la implementación del caso de estudio más las pruebas en la primera etapa.

Es destacable la imposibilidad de agregar la tag *<!DOCTYPE html>*, requerida para declarar que la página está en el formato Html 5, reportado actualmente como una falla en Links [48]. La ausencia de esta declaración podría llevar al navegador a interpretar el código como otra versión de Html y por lo tanto desplegar la página incorrectamente.

Tampoco en general es posible especificar un *Xml DTD* (*Document Type Definition*), tipo de datos Xml, definiendo la estructura de la página más allá de si es Xml, con por ejemplo el *tag html* como raíz. La actual generación de Html por lo tanto, permite generar Html no válido, con tag no válidas, u orden no válido. Esto puede ser visto como una mejora en la definición del tipo Html en Links, ya que exige que el tipo Html generado sea Xml estricto, pero no que tenga cierta estructura dentro de los Xml estrictos. O sea el chequeo de tipos de Html es mejor, pero no todo lo bueno que podría ser.

No es posible declarar atributos que tengan guiones en sus nombres, algo compatible con Html 5. En Html 5 los atributos personalizados deben comenzar por “data-“. Esto resultó en una limitante para el uso de Bootstrap, que necesita declarar estos atributos para usarlos en el menú. Es por este motivo que en el caso de estudio los menúes no se contraen (no aparecen solo tres líneas, que al tocar permite desplegar el menú) en dispositivos móviles.

Al elemento *<html>*, no es posible definirle atributos. Importante especialmente para la definición del lenguaje de la página, por ejemplo: *lang=”sp”*. No es posible hacer esto, salta un error de compilación. Tampoco se pueden definir el atributo *style* al elemento *body*.

No se pueden agregar comentarios Html: *<!--mi comentario -->*.

No se puede utilizar el carácter “&” para caracteres especiales en Html, como espacio.

Links genera automáticamente los *id* de los elementos *input* en una *form*, si usan *l:name*. Esto tiene limitantes importantes, no permite al desarrollador definirlos él mismo, y por lo tanto usarlos para poder manejar esos *input*. En particular no permite asignar o recuperar el valor de un *input* cualquiera en tiempo de ejecución, no se puede usar *label* con el parámetro *for*, para asociar una etiqueta con el *input*, etc. En caso de querer conocer el *id* de un *input*, se debe generar la página web, luego inspeccionarla, y así obtener el *id* del componente, luego se vuelve a modificar el programa con el *id* obtenido. Si cambia la página puede cambiar el *id* nuevamente.

Relacionado con el punto anterior está el uso de los *radio buttons*. En los requerimientos del caso de uso, estaba el caso del campo *sexo*, que tenía dos opciones. Se agregó con la idea de utilizar este componente. No fue posible, ya que para agregar *radio buttons*, se tienen que agregar varios *inputs*, uno por cada *radio button*, en este caso dos, y asignarle el mismo atributo *name*. El problema surge cuando es necesario usar el atributo de Links *l:name.* Al

compilar este programa, Links detecta dos input con la misma definición de *name* y no compila, no es posible lograrlo de esta manera, a pesar que es la manera típica de usarlo con Html. Se tuvieron que buscar alternativas, una era simplemente usar un *select*. Otra más elaborada, consistía en usar un *input* tipo *hidden*, y setear el valor de este campo cada vez que se hacía clic en los *radio buttons*. El *input hidden* debería usar el *l:name* buscado. Para usarlo habría que usar el *id*, que no está disponible hasta luego de compilar como se explicó previamente. Esta última opción no funcionó, al setear el campo *name* del *radio button*, y usar el atributo *l:onclick*, no hay efecto en el otro *radio button* y no se llama el código dentro del atributo *l:onclick*. Otra alternativa era usar *formlets*, pero esta opción no permite que los dos *radio buttons* estén no seleccionados a la vez, además de otros inconvenientes como se explicará en la sección correspondiente. Finalmente se optó por la más sencilla, que era usar el *select*.

En cuanto a la conversión de tipos, todos los tipos generados desde los *forms* son *strings*. Esto es consecuencia de que los *forms* son los *forms* de Html. Así por ejemplo si se quiere ingresar la cédula de identidad en el caso de estudio, no es posible especificar que el tipo devuelto debe ser *Int*. La conversión debe ser realizada directamente por el programador en caso de ser necesario algún procesamiento en el lenguaje Links. Por ejemplo si queremos validar que sea un número, debe primero usarse una expresión regular, y luego el conversor del lenguaje de *String* a *Int*. Lo mismo ocurre si es necesario definir un *formlet* personalizado (ver la sección *formlets*).

Algo similar ocurrió al implementar el campo fecha de nacimiento en la historia Agregar/Modificar Estudiante. La idea original era usar tres campos Int como aparecen en alguna documentación, y luego agregar validación sobre estos campos. Links tampoco dispone del tipo *Date* por lo que también si era necesario procesar la fecha, iba a ser necesario realizar una conversión a un tipo personalizado y también aparte implementar funciones de fechas para validar que sea correcta. Otra opción es indicar el atributo *type* en el input a *date* utilizando las nuevas capacidades de Html 5. Esto indica al navegador que utilice un componente adecuado para poder seleccionar la fecha. Además tiene como ventaja que la validación de la fecha la realiza el navegador. Otra ventaja es el formato del *string* representando la fecha que genera el navegador, que es compatible con el formato del mismo tipo fecha de muchas base de datos, *yyyy-mm-dd*. Finalmente se adoptó esta opción. En cuanto a la validación también se podría agregar que la fecha de nacimiento ingresada sea menor a la fecha actual. Para esto es necesario obtener la fecha actual, para lo cual existen funciones en Links que devuelven esta fecha, que es un variante con campos día, mes, año, hora, minuto y segundo. Esta función está fallando en la arquitectura Arm donde está desplegado el servidor de producción. Devuelve que el año actual es 1950. Sí funciona en Amd64 . Por lo tanto se decidió no implementar esta restricción. Otra opción sería corregir esta función. Esto se pudo haber realizado teniendo en cuenta las diferencias de arquitectura entre Arm y Amd64 pero dado que el objetivo de este proyecto es evaluar el lenguaje Links y no necesariamente corregirlo no pareció razonable esta opción. Este criterio se siguió en otras situaciones similares.

La capacidad de dividir una página en partes y ensamblarla, también tiene ciertas desventajas. No es posible lograr cierto nivel de abstracción, por ejemplo se podría querer

crear un componente que fuera un *input* de cierta forma, con un *l:name*, para poder reutilizarlo en distintas partes. Esto no es posible por limitaciones del lenguaje [2], lo máximo que se puede abstraer usando funciones es un *form*, que sí contenga elementos con *l:name*. Existe la posibilidad de usar *formlets*, que sí permiten este nivel de abstracción, pero tiene otras desventajas, se analizarán en su sección correspondiente.

En vez de usar la notación que incluye Xml directo en otro Xml, se puede incluir usando *page slices*, que permite corregir ciertos problemas [10]. Esta implementación incluye un *Page* dentro de otro *Page*, no un valor Xml dentro de otro Xml, como fue implementado originalmente en Links. Los *Page* a su vez son construidos a partir de un valor Xml. Pero tambień hay desventajas en esta implementación. Con un valor Xml se puede obtener un *Page*, pero con un *Page* no se puede obtener su contenido de Xml. Y todas las funciones de manejo del DOM, utilizan Xml en vez de Page, por lo tanto, si se quiere dividir la página en fragmentos usando Page, no se puede usar tan fácilmente Ajax, ya que va a ser necesario obtener los fragmentos Xml en vez de los Page. No basta con encapsular solamente los fragmentos page, es necesario también encapsular los fragmentos Xml.

En cuanto a Ajax, no fue posible usarlo, para la historia de inscripción a cursos del estudiante, donde se quiere inscribir o desinscribir a un estudiante de un curso, y luego por Ajax, refrescar la lista de cursos en la misma página. Hay errores o inestabilidades del lenguaje. La implementación de esta parte, incluye una *form* para cada botón, donde en el *l:onsubmit* de la misma, se llama a una función, que debe validar que el curso exista, inscribirse al curso y luego recargar por Ajax el listado. Lo que ocurre es que la aplicación recarga la página actual, sin realizar ninguna otra operación. Luego de investigar sobre el fallo, parece deberse a un error en *replaceNode*, después de llamadas a la base de datos. Debería funcionar correctamente, y que el sistema haga varios llamados Ajax, uno para la base de datos y maneje el DOM, pero no fue posible en esta implementación. *replaceNode* parece fallar en este contexto, ya que haciendo *print* justo antes del mismo se muestra su ejecución, o sea esa línea es llamada. Pueden verse estas pruebas en *pruebaCasoEstudioAjax.links*. Inestabilidades semejantes ya se habían comprobado en la primera etapa de pruebas del lenguaje, por ejemplo si se usa el sistema de módulos, en algunos casos da error de compilación, y se puede ver que esto desaparece simplemente al definir el alias de un tipo, que incluso no es más usado. Esto puede verificarse en *pruebas.links*. También para la falla asociada a *replaceNode*, parece muy relevante un *bug* reportado en el mismo lenguaje, *“replaceNode fails silently if the xml argument contains a l:action form”,* falla #219 [48]. Esta falla está asociada a *l:onclick*, cuando nosotros usamos *l:onsubmit*, pero parece de la misma naturaleza.

Por otro lado Links no provee características avanzadas como *two-way binding*. Actualmente existen frameworks como AngularJS [49], que proveen ayudas al modelado de páginas en Html. Utilizan por ejemplo *two-way binding*, que permite asignarle un modelo a un componente, y que este modelo sea actualizado en el cambio del valor del componente, y que un cambio en el modelo cambie el valor mostrado en el componente. Esto simplifica el desarrollo. Links no dispone de esta característica.

Otra carencia del lenguaje que aparece en la bibliografía [2], es que no es posible crear dinámicamente campos con *l:name*. Por lo tanto si en tiempo de ejecución es variable la cantidad de campos de una *form*, no será posible implementar este requerimiento.

Conclusiones

Así se puede ver como en este aspecto de Html, el problema de *type mismatch*, no está totalmente resuelto. No hay conversión de tipos automática desde Html, se sigue necesitando la conversión de tipos, si no se hace mediante Html, se tiene que hacer mediante *formlets* personalizados. Existe Html requerido para el funcionamiento de frameworks o el propio navegador que no es generado por Links. La expresividad de Links es limitada en este sentido.

Estos problemas llevan a deficiencias de factibilidad del lenguaje para satisfacer ciertos requerimientos. Por ejemplo, si se quiere que el menú se contraiga en dispositivos móviles, entonces hay que modificar Bootstraps, o Links, o usar otra alternativa. Esto dificulta la implementación de los requerimientos, y aumenta los tiempos de desarrollo. Si parte del requerimiento hubiera sido usar directamente la biblioteca, porque facilita el desarrollo web y los desarrolladores tienen experiencia en ella y por lo tanto disminuye los tiempos de desarrollo y mejora la calidad del producto, no hubiera sido posible. Esto debería haber sido totalmente compatible con Links, al usar solo Css. Algo similar ocurre con los *radio buttons*, la manera natural de implementarlos, sería con un mismo *l:name* análogamente a como se implementan los otros campos, o alguna otra posibilidad provista por el propio lenguaje. Esto no está disponible, no hay manera de hacerlo directamente. También si se usa la biblioteca auxiliar *formlets*, hay otros inconvenientes, como se verá en la siguiente sección. Así que se puede decir que no hay factibilidad en este punto. La búsqueda de alternativas aumenta los tiempos de desarrollo. Con Ajax simplemente errores en el lenguaje impidieron su implementación con ciertos recursos de tiempo. Con más tiempo, podría buscarse otra alternativa para el problema, como hasta corregir el error en Links. Respecto a este punto se está entre aumento de los tiempos de desarrollo y simplemente carencia de factibilidad del lenguaje. Con el tiempo invertido en investigar el problema, esto es carencia de factibilidad.
Por otro lado, las ventajas originalmente expuestas mejoran mucho la mantenibilidad, legibilidad, correctitud y tiempo de desarrollo.

Haciendo un balance, se puede decir que la mantenibilidad, legibilidad, correctitud mejoran mucho. Los tiempos de desarrollo en general mejoran, si lo que se quiere implementar es relativamente sencillo. En caso de requerimientos más complejos, aumentan los tiempos de desarrollo. En cuanto a la factibilidad, ocurre algo similar. Si los requerimientos son simples, es posible hacerlo, pero si son más complejos, como incluir Ajax, o *radio buttons* no seleccionados, no es posible.

5.5 Formlets

Formlets es una biblioteca auxiliar para manejo de componentes. El uso de los mismos tiene varias ventajas. Se pueden reutilizar, logrando en principio código más legible y

mantenible y mejorando los tiempos de desarrollo, si se usa muchas veces el mismo componente. También manejan automáticamente la conversión de tipos desde Html a Links, devuelven un valor Links, por lo que no es necesario realizar la conversión. Pero además vienen con varias limitaciones e inconvenientes:

a. No es posible usarlos con Ajax, siempre hacen *submit*. El manejador del *formlet*, es una función que recibe los datos generados por el mismo, y luego devuelve siempre, una página web. Por lo tanto siempre se navega hacia una nueva página web. Esta es una limitante muy importante en la programación web actual, ya que muchas veces es un requerimiento no recargar toda la página. Le quita factibilidad a su aplicación.
b. Otro inconveniente es que todos los mensajes de error están *hardcoded*, entonces no es posible modificarlos, a menos de extender o reescribir el componente. O sea hay que volver a crear el componente de la biblioteca. Esto se puede ver en el ejemplo “formlets operaciones” de los casos de prueba web, basta no rellenar un campo. No se pueden usar estos componentes de la biblioteca, aumenta tiempos de desarrollo, para crear otro, mientras se utilice pocas veces.
c. También en cuanto a los mensajes de error, por ejemplo de Int obligatorio, están siempre junto al componente. Si un requerimiento es ponerlos como subtítulos, o al final del formulario, no es posible hacerlo, es necesario reescribir el mismo. O sea que también obliga a volver a crear el componente, hay que crear una biblioteca personalizada. También aumenta tiempos de desarrollo.
d. Otra característica es que si en una *form* se decide usar un *formlet*, también los demás elementos deben ser *formlet*, y hay que componerlos y usar todo en un *formlet* compuesto. Esto es debido a que hay un único manejador del *formlet*, que hace el submit y carga otra página, y los demás elementos del *form* deben estar en su contexto para poder ser procesados. Esto involucra un riesgo importante, si un elemento no puede ser representado por un *formlet*, entonces hay que deshacer todo y volver a implementar todos los elementos sin *formlets*. También a este nivel quita legibilidad, ya que para saber qué hace un *formlet*, hay que buscar cada uno de sus componentes, y analizarlo. Lo mismo con cualquier función de validación. Involucra cierta rigidez de diseño que compromete tiempo de desarrollo y factibilidad.
e. Hay complejidad en los *formlets*, lo que lleva a aprender otra sintaxis más, lo que no debería ser necesario a priori. Afectan tiempo de desarrollo, si se usan pocas veces.
f. En caso de querer crear un *formlet*, en principio, se deben usar a su vez otros *formlets* básicos. A veces esto no es suficiente, como en el caso de *radio buttons*. En este caso deben crearse de cero. Crear de cero un *formlet* es complejo, y no está documentado. El desarrollador se introduce en el código fuente del lenguaje. Se puede ver inspeccionando este código fuente, que se utilizan ciertas funciones no documentadas, y operadores no documentados. Es destacable notar también que, en caso de crearse un formlet de cero, es necesario agregar toda la conversión de tipos que sea necesaria para el componente, desde el String de Html al tipo de Links apropiado. Afectan tiempo de desarrollo.

En cuanto al uso de *radio button* para el campo sexo como se comentó en la sección Html-Css, se podrían usar *formets* para este propósito. Lamentablemente como se puede ver en las pruebas web, este componente tiene un error, si no se selecciona ninguna

opción, lo que es razonable al principio, para obligar al usuario a elegir una, produce un error interno al hacer submit. Este es un error que no vuelve factible su uso. No es posible usar otro componente tipo *radio button* más básico, este es el más pequeño posible. Esto lleva a la necesidad de reescribir el componente. También esto lleva a que todos los demás componentes sean *formlets*, incluso el campo fecha, con todos las desventajas nombradas, además de otros posibles problemas como el anterior. Por lo tanto no se eligió esta opción.

Además en cuanto al caso de estudio, son relativamente pocos componentes y no es necesaria mucha reutilización, lo que reafirma la decisión tomada. La misma decisión se tomó en un trabajo relativamente reciente, implementado por los propios desarrolladores del lenguaje [9].

La no capacidad de los formlets de funcionar con Ajax, se podría decir es crítica actualmente y casi definitoria en cuanto a su utilización. Si se agrega la característica de tener que convertir todos los componentes a *formlets*, esto lleva a que su uso sea restringido para casos de muchos componentes repetidos, y sin Ajax. Pruebas sobre *formlets* pueden verse publicadas junto al caso de estudio en pruebas web.

Junto a las conclusiones de la sección Html-Css, se llega a que si un requerimiento es utilizar Ajax, Links no posee factibilidad de satisfacer ese requerimiento.

5.6 Javascript

En cuanto Javascript, Links tiene importantes ventajas.

Ventajas

Es conocida la facilidad con que se cometen errores en Javascript, una primera consecuencia de su tipado débil. Es posible escribir código como:
e = e || {};

el operador || hace una conversión de tipos que dificulta la interpretación de este resultado. Si *e* es *undefined* devolverá *{}*, si es 2 devolverá 2. Esto lleva a posibles errores de interpretación.
Otro problema relacionado es que el código Javascript no puede ser comprobado en cuanto a sintaxis a menos que sea ejecutado en un navegador. Esto enlentece el desarrollo y afecta especialmente la correctitud, hasta que no corre no es posible comprobar el código.

Links no tiene este problema. Todo el código se puede interpretar más fácilmente. El tipado fuerte detecta en tiempo de compilación muchos errores. Lo mismo ocurre con la sintaxis, no es necesario que esté corriendo en el navegador para comprobar el código. Esto acelera los tiempos de desarrollo y mejora la correctitud. Este resultado también es observado en otros lenguajes funcionales con similares características de tipado fuerte y compilación, como Elm [50]. Tambień al encontrarse más fácilmente los errores, mejora mantenibilidad.

Limitaciones

A pesar de las grandes ventajas con respecto a Javascript, Links también posee limitaciones.
No posee capacidad para minimizar código Javascript. Esto lleva a que el código Javascript generado sea de importante volumen, del orden de cientos de KB por ejemplo cuando una biblioteca Javascript típica es del orden de decenas de KB. Esto aumenta los tiempos de envío de la página, lo que es muy importante actualmente.

También hay errores en ciertas funciones Javascript, como *replaceNode*, como se mencionó en la sección Html-Css. Otros errores pueden encontrarse en las pruebas web:

- Por algún motivo si se escucha por las funciones *l:onmousedown* y *l:onkeydown* no se puede escribir en su input, lo que no pasa con las otras funciones.
- Al escuchar el evento *l:onload*, como atributo del *body*, como es esperable en Html, hace que Links muestre en la página web un mensaje: *key="11"* donde *11* puede ser 2, 6, etc. al refrescar la página. Parece un mensaje de pruebas del propio lenguaje. También no ejecuta la llamada a función que hace el *onload*, o sea no funciona.

También no están disponibles muchas funciones Javascript*,* como *submit,* que permite hacer un *submit* de una *form***.** Esta función ayuda a dinámicamente enviar una *form***.**

Conclusiones

Como se puede ver, las ventajas de Links en cuanto a Javascript son muy grandes, el tipado fuerte, junto con la compilación, permiten importantes mejoras en legibilidad, mantenibilidad, y especialmente en correctitud. Por otro lado, incluye problemas de performance junto con que no equivale en expresividad a Javascript. No se logra resolver completamente el *type mismatch problem*. Hay muchas funciones en Javascript que no están presentes en Links, y otras tienen errores que llevan a que no sea posible utilizarlas. Esto finalmente afecta la factibilidad de ciertos requerimientos. Aunque no es nuestro caso, si un requerimiento implicara el escuchar por uno de los eventos del *mouse* no disponibles en el lenguaje, simplemente no sería posible satisfacerlo.

Por lo tanto nuevamente, si estamos con una aplicación simple será posible realizarla en Links, por el contrario, si es más compleja, no será posible realizarla. También si un requerimientos es que sea lo más ágil posible en la descarga (performance), este se ve afectado seriamente por la implementación de Links.

5.7 Cookies y persistencia en el cliente

La persistencia por el lado del cliente también es un punto importante actualmente. En el caso de la implementación del login en el caso de estudio, se utiliza para guardar información de un login exitoso. Esta característica está fuertemente limitada en Links, no es posible asignar una fecha de expiración a un cookie, además de no disponer de las

nuevas funcionalidades de Html 5, como *sessionStorage* y *localStorage*. Así por ejemplo no es posible establecer sesiones basadas en pestañas del navegador, sino que todas están basadas en el navegador en sí. Esto es debido a que en cualquier pestaña del navegador bajo cierto dominio, se tiene exactamente el mismo cookie. Esto no ocurre con *sessionStorage*, con él la información es por pestaña, así, sería posible establecer sesiones diferentes en distintas pestañas [55].
También se observa que al cerrar el navegador, inmediatamente se pierde sesión. Esto se debe a la ausencia de posibilidad de asignar una fecha de expiración a la cookie. Otra característica de *sessionStorage* y *localStorage* es que pueden almacenar más información que una cookie.

Pueden encontrarse pruebas sobre estos puntos con cookies en la aplicación en la sección pruebas web.

La imposibilidad de sesiones por pestañas, más el no poder evitar el cierre de sesión al cerrar el navegador, habla de la carencia de factibilidad del lenguaje en caso de requerirse estas características.

5.8 Continuaciones

Las continuaciones son una característica especialmente destacable de Links, no disponible en muchos lenguajes imperativos. Por ejemplo, existen bibliotecas en Java, pero están obsoletas [51]. Las continuaciones son facilitadas por el estilo de implementación del propio lenguaje Links, mediante *continuations-passing style* [2, 52]. A su vez, la implementación de Links mediante *continuations-passing style,* es facilitada por la característica funcional de Links, a diferencia de los lenguajes imperativos [52].
Las ventajas de Links al poseer esta característica son muy relevantes, dado un programa y un estado del mismo, se puede volver a ese punto fácilmente, lo que permite implementar con relativo bajo costo requerimientos que en otros lenguajes son mucho más elaborados.

Como se mencionó en la sección continuaciones de la introducción al lenguaje, destaca especialmente el uso del login, pero también el manejo de errores (como el manejo de excepciones mediante efectos), pantallas de confirmación, etc.

Típicamente, en Java con Spring [53], si el acceso a una página no es permitido, o se perdió la sesión, se navega a una página de acceso no autorizado, o de login. Desde la página de login, se navega a la página principal de la aplicación. Toda la información previa al login se pierde, se pierde por ejemplo cualquier información en cualquier formulario que se estaba completando.
Esto no ocurre en Links, no se pierde la información previa. En caso de perderse sesión, se navega a la página de login, pero a partir de allí se continúa con la operación pendiente, o sea con el envío de los datos del formulario, no es necesario reingresar nada nuevamente.

También una de las implementaciones más comunes de la seguridad (que el usuario esté autenticado por ejemplo) en Java es mediante interceptores [53]. Esto lleva a que se pierda

legibilidad del código, no es fácilmente visible donde se comprueban las credenciales para permitir el acceso a la página. Esta comprobación ocurre en un código ajeno a la ejecución de la página.
Esto tampoco pasa en Links, es fácilmente visible donde se hace la llamada, y es posible entender más fácilmente el código.

Por todo esto, las continuaciones presentan grandes ventajas de tiempo de desarrollo, mantenibilidad, legibilidad y correctitud, sobre la programación web tradicional.

5.9 Interfaz con el exterior

Para cualquier lenguaje, especialmente uno que se está desarrollando, es especialmente importante proveer interfaces con otros programas o servicios. La comunicación con el exterior puede ser crítica, en caso de no disponerse de una funcionalidad, esta podría resolverse mediante alguna biblioteca externa, programa o servicio.

La comunicación de Links con el exterior se da solamente en lo que se correspondería a capa presentación, y capa de persistencia. Solo es posible mediante Html-Javascript, con eventos de usuarios, por ejemplo, y a través del acceso a la base de datos. No dispone de otras interfaces.
Se destaca notablemente la carencia de biblioteca de entrada/salida. No es posible realizar operaciones sobre el sistema de archivos, leer archivos, modificarlos, guardarlos. La ausencia de esta característica llevó a que en la necesidad de este requerimiento, se tuviera que modificar el propio lenguaje, creando una rama del mismo, en colaboración con los desarrolladores, como se puede ver en [7]. Estos cambios no se incorporaron a la versión oficial del lenguaje.
También es destacable la imposibilidad de interacción con el sistema operativo, no se pueden ejecutar programas nativos. Esta es en general una alternativa de último recurso para resolver requerimientos no compatibles con el lenguaje, pero no está disponible en el mismo. Un ejemplo es procesamiento de imágenes, donde se requiere alta eficiencia y especialización. Una aplicación web que requiera uso de mapas personalizados, podría tener este requerimiento.
Tampoco se puede por ejemplo interactuar con un servidor de correo. Si un requerimiento del login hubiera sido enviar correo de confirmación de alta de usuario, lo que es muy común, esto no habría sido posible.

En nuestro caso de estudio, esta limitación de la interfaz con el exterior afectó el requerimiento de mostrar la temperatura en Montevideo. Este requerimiento fue pensado para evaluar justamente la posibilidad de interacción del lenguaje con el exterior. No fue posible implementarlo. Links en sí mismo sólo posee las interfaces que se nombraron, no se pueden consumir servicios web. Se evaluaron alternativas. Links posee la posibilidad de incorporar Javascript nativo en las páginas web, aunque rompe el espíritu del lenguaje, al usarse otro lenguaje para resolver un problema, se analizó esta opción.

Se quiere acceder a:

http://samples.openweathermap.org/data/2.5/weather?q=London,uk&appid=b6907d289e10d714a6e88b30761fae22
que brinda un servicio web de temperatura, devolviendo un objeto en formato Json.

El primer problema fue que Links daba error de compilación al usar el carácter “&”, en un javascript pidiendo por Ajax este recurso. Como se comentó en la sección Html-Css, no es posible usarlo para los caracteres especiales de Html, y tampoco dentro de Javascript para una url.
Dada esta situación, se optó por utilizar un archivo .js, servido estáticamente por el servidor. Este archivo contenía el código Javascript que no compilaba en Links. Al ser servido estáticamente, se saltaba cualquier chequeo del lenguaje Links. Así se implementó una función Javascript que es llamada dentro del html de la página Links.

Lamentablemente, tampoco dio resultado, la razón es la característica de seguridad CORS de los navegadores [54]. La idea es que por defecto, una página web puede consumir servicios solamente desde donde fue servida. En nuestro caso, el dominio de origen es *abentan.ddns.net*, por lo tanto, por defecto si la página web de listado de cursos por ejemplo quiere consumir un servicio web de cualquier dominio distinto de éste, el navegador lo impedirá, mostrando un error en la consola Javascript. Esto es lo que se puede ver en el ejemplo “Javascript puro” de los casos de prueba web. Se muestra este mensaje de error en la consola javascript accesible por F12 en Google Chrome:

Failed to load http://samples.openweathermap.org/data/2.5/weather?q=London,uk&appid=b6907d289e10d714a6e88b30761fae22: Redirect from 'http://samples.openweathermap.org/data/2.5/weather?q=London,uk&appid=b6907d289e10d714a6e88b30761fae22' to 'https://samples.openweathermap.org/data/2.5/weather?q=London,uk&appid=b6907d289e10d714a6e88b30761fae22' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:8080' is therefore not allowed access.

Pero hay una manera de permitir ese acceso a servicios fuera del dominio del servidor de la página web original. La página web original, debe ser servida, con ciertos *headers* especiales, en la respuesta Http, para indicarle al navegador, que se relaja esta restricción de seguridad, y se permiten ciertos otros dominios además del dominio original.
O sea cada vez que se sirve una página web, debe ser servida con cierto *header*, este *header* es *'Access-Control-Allow-Origin'.* Se debe modificar el modo de servir páginas del servidor web Links para permitir esto. Esto lamentablemente no es posible, no hay manera para el programador de hacerlo en Links.

Hay otras alternativas, como crear algún servicio web en el mismo servidor, en otro lenguaje, para evitar este problema. También se podría crear alguna aplicación en otro lenguaje, para consumir el servicio web del clima, y luego almacenarlo en la misma base de datos que usa Links, para que sea accesible por el lenguaje. Estas alternativas no aportan a la evaluación del lenguaje, y son muy costosas.

En resumen, puede verse como en este aspecto Links es muy limitado. Hay serias limitaciones en el propio lenguaje, que llevan a la no factibilidad de implementación de variados requerimientos. La utilización de Javascript lleva a claramente no trabajar en el propio lenguaje, sino en otro en forma auxiliar, lo que incluso si se encontrara una solución mediante su uso, comprometería muchas de las ventajas del mismo, se vuelve al problema de *type mismatch*, por lo menos parcialmente. Pero incluso mediante su utilización, no es posible consumir servicios web. Esto conduce a la no factibilidad de Links en satisfacer este requerimiento del caso de estudio, y cualquiera de los nombrados, los cuales son muy comunes en la programación web actual. No se pueden consumir servicios web desde Facebook, Tweeter, etc.

5.10 Manejo de errores

El manejo de errores en todo lenguaje es importante, desde los errores que se produzcan por operaciones propias del lenguaje, como una división por 0, hasta errores personalizados por el programador que permitan un mejor manejo de situaciones excepcionales. No es lo mismo que todos los errores deban mostrarse sí o sí en salida estándar, a que existan un manejo de excepciones avanzado por ejemplo. Veremos qué alternativas provee Links para este propósito.

Errores de funciones nativas del lenguaje

En cuanto al manejo de errores propios del lenguaje, Links no permite que sean manejados por el programador, solo muestra un mensaje de error en consola, y además a veces en la página web, dependiendo de si el programa corre con servidor web o no. Por ejemplo, en caso de división por 0, muestra en consola:

...
Re-raised at file "camlinternalLazy.ml", line 34, characters 4-11
Called from file "core/errors.pp.ml", line 114, characters 4-16
**** Error: Division_by_zero*

En caso de conversiones, si falla *stringToInt* en una aplicación web, una función parcial aplicada a “a” por ejemplo, devuelve la siguiente página:

Figura 5, mensaje de error en caso de falla de *stringToInt*.

y además este mensaje en la salida estándar:
int_of_string

Sin ninguna información de dónde se originó el error, como número de línea.

También pueden darse errores en tiempo de ejecución en la base de datos, si los tipos de datos asignados en la definición de la tabla en Links no se corresponden con los datos de la tabla en la base de datos, por ejemplo. En este caso, si la aplicación es web, también muestra el error directamente como página web:

Figura 6, mensaje de error en caso de falla en la base de datos.

Otro ejemplo de error con una consulta es:
Links Fatal Error
Bad base value: `For ([(2632, `Table ((..., "postgresql:casoestudio::5432:usuario:"), "usuarioscursos", [], ({"idcurso" => `Present `Primitive `Int ; "idusuario" => `Present `Primitive `Int ; }, ..., false)))], [], `If (`Apply ("&&", [`Apply ("==", [`Project (`Var (2630, {"id" => Int; "nombre" => String; }), "id"); `Project (`Var (2632, {"idcurso" => Int; "idusuario" => Int; }), "idcurso")]); `Apply ("==", [`Project (`Var (2632, {"idcurso" => Int; "idusuario" => Int; }), "idusuario"); `Constant `Int 1])]), `Singleton `Record {"idusuario" => `Project (`Var (2632, {"idcurso" => Int; "idusuario" => Int; }), "idusuario"); }, `Concat [])

En los dos casos también se muestra el correspondiente mensaje en la salida estándar.

Como se puede ver, la estrategia de manejo de errores en Links, es simplemente mostrar el error en una nueva página web, y además en la salida estándar. No provee alternativas para el manejo de sus propios errores, como podrían ser mónadas Maybe, o excepciones. Esta estrategia tiene sus ventajas, si sabemos que el error no va a ocurrir, por alguna precondición, el código queda más legible, ahora tiene especialmente desventajas. Para evitar que salte el error, es necesario previamente a la invocación de la función, hacer un chequeo de los parámetros de la misma, para asegurarnos que no ocurra. Por ejemplo en cuanto a la conversión de String a Int, necesaria en la historia Agregar Estudiante, para convertir la cédula a Int, es necesario usar expresiones regulares para asegurarnos que es un entero lo que se ingresa. Esto con respecto a la alternativa de excepciones tiene la

desventaja de que muchas veces es necesario implementar manualmente estas comprobaciones. Más aún, a veces simplemente no es posible hacerlo como en el caso de la base de datos. Previamente no se pueden evaluar ciertas situaciones, como el mapeo incorrecto de tipos.

Todo esto lleva a que se muestren mensajes inadecuados para el usuario, afectando la calidad de la aplicación. Si un requerimiento implicara evitar estos mensajes, no podría implementarse en Links. Esto afecta la factibilidad del lenguaje.

Por otro lado, es destacable una observación, Links sí dispone, en el uso de *session types*, de un símil *try-catch* de Java. O sea sí se dispuso de excepciones, para el manejo de errores en *session types*. ¿Por qué no se utilizó en general? Esto se responderá en la sección “un lenguaje experimental” en las conclusiones.

Errores en funciones creadas por el programador

El programador en Links, tiene la posibilidad de implementar varias estrategias de manejo de errores, para las funciones creadas por él mismo.
Por supuesto la más sencilla es la implementada en Links en sus funciones nativas, simplemente se muestra el error en la salida.
La siguiente ya nombrada es el uso de un tipo Maybe, lo que parece muy útil en general, la función ‘envuelve’ su valor devuelto en un variante, que retorna *Just x* si la función devuelve x, y *Nothing* o algún otro valor si se produjo un error.
Otra alternativa nombrada en la introducción del lenguaje, es mediante el uso de efectos, incluso se planteó un ejemplo con manejo de excepciones. A pesar de las grandes posibilidades que brinda esta opción en general, incluso con continuaciones, también tiene sus inconvenientes. El primer ejemplo mostrado en la introducción era:

```
sig myDiv : () {Fail|_}-> Int
fun myDiv() {
  var x = 1;
  var y = 0;
  if (y == 0) {
    do Fail;
    0
  } else {
    x/y
  }
}

# manejador de la excepción Fail, en cuyo caso devuelve Nothing
handler maybeResult {
  case Return(x) -> Just(x)
  case Fail(_) -> Nothing
}
```

La función que invoca a *myDiv*, va a tener que usar un *handler*. Este *handler*, solo devolverá un nuevo variante, con un valor especial para el caso de la excepción. No se da una situación similar al *try-catch* de *session types* o Java, donde la función invocante de en nuestro caso *myDiv*, puede procesar ella misma esta situación de la excepción, con código específico. O sea en cuanto al manejo final, no hay mayor diferencia que con el usar mónadas desde un comienzo.
También se presenta otra limitante importante: los *handlers* solo aceptan funciones que no tienen parámetros. No fue posible implementar en nuestras pruebas *handlers* que manejaran funciones con parámetros. Por lo que su uso es también limitado a este respecto.

Otro aspecto, comparado con Java, es que en Java, hay dos tipos de excepciones, *checked* y *unchecked* [56]. Las tipo checked, obligan al programador a manejarlas explícitamente, las tipo unchecked, pueden no ser manejadas, y en caso de saltar se mostrará la excepción. Esto permite código más legible, en caso de no querer o ser necesario manejar la excepción explícitamente, y manejo de excepciones explícito en caso por ejemplo de no poder chequearse una precondición. Esta característica no está disponible en Links.

Así vemos que en cuanto a la posibilidad de manejo de errores por parte del programador, Links presenta limitantes. No es posible usar *handlers* con funciones que tengan parámetros. Incluso en este caso, el valor devuelto por el *handler* es una mónada, o eventualmente otro valor. Esto comparado con otros lenguajes que presentan la posibilidad de manejo mediante *try-catch*, implica para Links en este aspecto, menor legibilidad, y por lo tanto menor mantenibilidad. También el hecho de buscar estas alternativas para manejo de excepciones, implica un mayor tiempo de desarrollo.

5.11 Información de errores

Dadas las particularidades de Links respecto a la información en caso de error, es importante analizar al menos brevemente este aspecto. En la sección previa, se señalaron varios errores y sus mensajes de error, representativos del lenguaje. Podemos agregar este para ilustrar un poco más:

```
...
Called from file "core/errors.pp.ml", line 114, characters 4-16
:0: Type error: The function
    `<dummy>'
has type
    `(() ~a~> Page) ~c~> String'
while the arguments passed to it have types
    `() ~a~> Int'
and the currently allowed effects are
    `wild|d'
In expression: <dummy>.
```

Este error ocurre en tiempo de compilación, si dentro de un elemento *form*, en su atributo *l:action*, se devuelve un valor que no es tipo *Page*. Como se puede ver, este error muestra como último valor el número de línea interno, de la implementación en sí de Links en O'Caml. No muestra el número de línea de donde ocurrió en el programa escrito por el desarrollador. También la descripción del error no brinda más información al desarrollador. En estos casos debe buscarse el error dividiendo el código en partes, simplificándolo sucesivamente, hasta encontrarlo.

Como podemos ver, muchas veces los mensajes de error del propio lenguaje no brindan información al desarrollador, o la información es difícil de interpretar. Esto lleva a que sin experiencia en el lenguaje Links, los tiempos de desarrollo se vean afectados por esta característica.
A conclusiones similares se llega en [6].
Este punto da pie a la conclusión de que el lenguaje no ha llegado a un nivel de madurez suficiente.

5.12 Documentación

Otro aspecto del lenguaje Links es su documentación.
La documentación en Links se encuentra dispersa, e incompleta. Para entender ciertos aspectos del lenguaje, es necesario leer *papers*, que no están pensados para introducir a un programador al lenguaje, y manejan conceptos que están más allá del mismo. Un ejemplo es efectos, otro es el sistema de tipos. Para entender el sistema de tipos, es necesario tener conocimientos de *row polymorphism*, no presente en la documentación, sino solo nombrado. Lo mismo ocurre con otro aspecto muy nombrado como cálculo lambda. Muchas funciones no tiene documentación, como *there*, y no se dispone de ninguna ayuda en línea, solo un listado. Este listado a través de @builtins, está además incompleto, no incorpora toda la parte de eventos web.
Un primer ejemplo claro es el manejo de Strings, que se encuentra mal informado en la documentación [1]. Hubo que revisar @builtins, a través de tipos de entrada y salida, y luego ejemplos del lenguaje, para poder dar con la función concatenación de Strings, y otras como substrings. Esto al final lleva a que la manera de conocer qué hacen estas funciones es a través de experimentación, y a través de exploración del código fuente.

No existe ayuda de las opciones del comando *linx*, que ejecuta programas Links, sus opciones están desactualizadas en la documentación (como -O), o ausentes en muchos casos (como -m o --path). Para obtener información de las mismas, se debe explorar la wiki, o información de liberación de versiones del lenguaje.
Lo mismo ocurre con las opciones de configuración. No existe ayuda en línea de las mismas, ni documento centralizado. En general en cuanto a configuración, se debe explorar los 3 documentos anteriores para poder encontrar información.

También se observó que la inclusión de la documentación no está presente en el proceso de desarrollo del lenguaje [30].

Esta falta de documentación, ha afectado el desarrollo de ciertos proyectos, como [7]. Allí no sabían que existía cierta función, e implementaron por otra vía lo que Links ya tenía.

Asociado a la documentación, sí están disponibles muchos ejemplos que muestran como resolver ciertos problemas. Estos ejemplos resultan especialmente útiles, desde que dados ciertos problemas, se pueden buscar las soluciones que los propios desarrolladores eligieron en el lenguaje, las cuales deberían ser las mejores. También tienen como limitación natural que estos ejemplos no cubren todos los casos posibles, pero sí una variedad interesante. Además no tienen muchos comentarios incluidos. Están disponibles en [8].

Las conclusiones con respecto a la documentación del lenguaje, son que para alguien sin experiencia en Links, hay dificultades para comenzar a programar en el lenguaje, especialmente en características ajenas a los ejemplos. Esto afecta los tiempos de desarrollo, en el comienzo de un nuevo proyecto.

Debemos hacer notar que este aspecto, de la programación en Links, es fácilmente corregible, y no destaca como un aspecto central en la evaluación del lenguaje.

5.13 Sistema de tipos

Merece una mención especial el sistema de tipos de Links. Links es fuertemente tipado, ya se comentaron las amplias ventajas que esto conlleva. Este sistema está implementado sobre inferencia de tipos. Esta inferencia permite que el programador no declare el tipo de cada función, sino que el mismo sea detectado por el lenguaje. Esto en principio conlleva ciertas ventajas, lo que acelera el desarrollo por lo menos en pequeños programas. Aquí interesa destacar especialmente otras características del sistema de tipos como los efectos y *row polymorphism*.

Como se pudo ver en la introducción al lenguaje, los tipos generados pueden ser muy complejos, especialmente por el uso de estos efectos, y el nombrado polimorfismo de filas. A este respecto Links dispone de una notación simplificada, para hacer más amigable el uso del mismo, como ~> para indicar el efecto *wild*. Incluso con esta notación, puede ser complejo, especialmente con los handlers de los efectos, un tipo de un handler puede ser:
fun : (() {Fail:_|b}~> c) -> () {Fail{_}|b}~> [|Just:c|Nothing|_|]

En este caso, la sola inferencia de tipos no es suficiente y es preferible usar alias de tipos junto a firmas de funciones para poder entender mejor el tipo de una función.

Incluso con un relativamente simple *map*:
links> map;
fun : ((a) -b-> c, [a]) -b-> [c]

ocurren situaciones similares, aquí aparecen variables representado efectos.

También aspectos como los efectos y polimorfismo de filas, no son comunes para el programador web promedio, e implican un aprendizaje.

La unión de aspectos nuevos como efectos más polimorfismo de filas, que implican tiempo de aprendizaje, junto con la complejidad que los mismos tienen, hacen que el sistema de tipos en Links sea mucho más complejo que otros, como en Java. Esto involucra menor legibilidad, y por lo tanto menor mantenibilidad, especialmente a medida que crece el producto y las funciones son más complejas.

De todas maneras, a pesar de que vale la pena ser nombrado, se considera un aspecto de menor peso en la evaluación del lenguaje.

5.14 Testing

Links no posee bibliotecas que faciliten las pruebas de los programas, como puede ser muy común actualmente, especialmente debido al auge del desarrollo guiado por pruebas (TDD por sus siglas en inglés). Especialmente no dispone de *stubs* sobre la base de datos, que permita realizar pruebas unitarias en funciones sobre la misma base de datos. Sí se pueden implementar fácilmente otros tipos de *stubs*, y utilizarlos en este tipo de pruebas. En nuestro caso en la primera etapa de realización de pruebas sobre el lenguaje, implementamos algunas funciones auxiliares muy sencillas con este propósito. Con estas funciones se pueden implementar muchas pruebas unitarias y de integración. Al ser las aplicaciones en Links de tipo web, pueden realizarse pruebas funcionales en las mismas con herramientas comunes de testing web, como por ejemplo Selenium [57].

La naturaleza funcional del lenguaje, facilita ampliamente el desarrollo de estas pruebas. La carencia de bibliotecas anteriormente nombradas, no nos pareció una característica que limite el uso de pruebas en el lenguaje. No se ve afectada la capacidad de correctitud a este respecto.

5.15 Performance

Este es un requerimiento no funcional presente siempre implícitamente, en todo desarrollo. Todo usuario quiere que la aplicación responda lo más rápido posible, o por lo menos que no tarde demasiado. En este trabajo no evaluaremos especialmente este aspecto, por lo que nos limitaremos a dar una evaluación general muy básica de los resultados obtenidos, además de presentar la información de otros trabajos que sí nos parecen relevantes.

La aplicación en dispositivos de escritorio se despliega con fluidez, y responde con velocidad típica a la interacción con el usuario. Esto lo comprobamos simplemente utilizando la aplicación repetidamente en cada funcionalidad implementada. Lo mismo ocurre con dispositivos móviles, no notamos mayor diferencia con otras páginas web. En este punto es importante notar, que se comprobó usando como servidor un Intel Core I5 a 3.0 Ghz y 8GB de memoria, no se usó el Raspberry Pi 3, de 1.2 GHz y 1GB de memoria, donde está el servidor de producción, que es notablemente más lento. También que esta es

un medida subjetiva por usuario, no se hicieron mediciones precisas de tiempos de respuesta.

En cuanto a otros trabajos, hay varias observaciones que aportan para evaluar la performance en Links. Links es un lenguaje interpretado, de donde es esperable que tenga menor performance que lenguajes compilados, o por lo menos con algún tipo de compilación intermedia como Java. Esto se aprecia en trabajos como [9], donde para una aplicación de gran volumen, se indica esta causa como un limitante importante en Links. Otro trabajo muy relevante a este respecto es [20], donde incluso se construye un compilador para el servidor, con ciertas limitaciones. Nos parece especialmente relevante presentar algunos de sus resultados. En la siguiente figura pueden verse dos gráficos, uno que mide la performance para construir y calcular la suma de los primeros n enteros, y otra para realizar un quicksort.

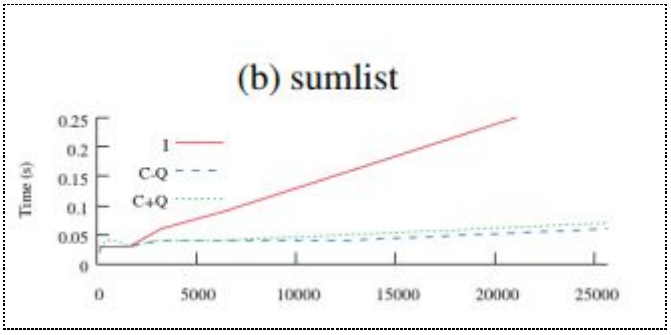

Figura 7, cálculo de la suma de n primeros enteros [20].

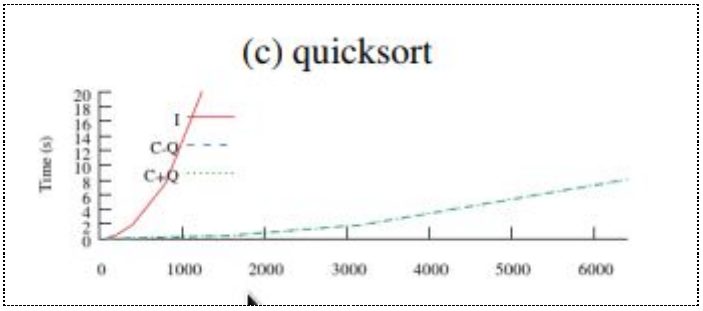

Figura 8, tiempos en el cálculo de quicksort [20].

En el gráfico, *I* se refiere al cálculo con Links interpretado.
C-Q se refiere al uso de Links compilado, sin implementar traducción a SQL de consultas a la base de datos.
C+Q es Links compilado y además implementa la traducción de consultas a la base de datos.
Se puede ver como la sola compilación de Links mejora su performance en un orden de magnitud de 1 o 2, lo que habla de su muy importante lentitud especialmente para aplicaciones que sean muy intensivas en computación.

También la implementación de efectos es poco performante. Se puede ver según [21] como la implementación de un compilador de efectos mejora sustancialmente este aspecto. En la

siguiente tabla presente en este trabajo, se puede observar la gran variación de tiempos entre Links interpretado, compilado y usar OCaml nativo para resolver distintos problemas.

	State	8-Queens	20-Queens
Links Interpreter	76167	242	411517
Links Compiler	1619	1	1059
OCaml (native)	829	1	200

Figura 9, tiempos de solución de ciertos problemas clave en Links [21].

Otro aspecto a notar en cuanto a performance, es la performance respecto a Javascript. Links implementa todas las funciones Javascript en un *continuation passing style*, para permitir continuaciones, muy diferente a código Javascript directo. Esto implica una pérdida de performance del un orden de magnitud según [2].

También en la sección acceso a la base de datos, hablaremos de performance en este aspecto, y veremos que algunas implementaciones son muy poco performantes.

Así podemos ver que la performance es un punto débil en la actual implementación de Links. En general, cualquier aplicación que requiera una respuesta óptima, implicará la no factibilidad de Links para la misma.

5.16 Responsive

La característica *responsive* es un requerimiento no funcional del caso de estudio. Como se ha comentado, se implementó a través del *framework* Bootstrap. Como regla general, podemos decir que fue exitosa, aunque hubo algunas pequeñas limitaciones como la no capacidad de contracción del menú para dispositivos móviles. Esto fue consecuencia de no poder definir ciertos atributos a un elemento Html, los que contengan el signo “-”, una limitación de Links, ya que no compila cuando está presente este signo en el nombre de un atributo. Como no se consideró de gran impacto esta situación para el usuario, junto con que solucionarlo pareció costoso, ya que habría que modificar el framework y probarlo totalmente, o implementar de cero el menú, no se buscaron más alternativas. Se comprobó el despliegue correcto de la aplicación en celular de 5.5” de diagonal de pantalla, y distintos anchos de ventana en dispositivos de escritorio. Todos fueron satisfactorios.
Sí se desprende de este requerimiento, como se comentó en la sección Html-Css, que si se pide por parte del cliente el uso de frameworks como Bootstrap, las características de Links pueden llevar a que no se puedan satisfacer, y por lo tanto, indicar la no factibilidad de Links en este aspecto.

5.17 Seguridad

La seguridad es otro requerimiento no funcional, actualmente de gran relevancia, implícito en casi toda aplicación web. Links no posee un framework de seguridad, pero son notables

algunos trabajos que brindaron algunas soluciones y expusieron muy serias carencias a este respecto.

Como se analizó previamente, las continuaciones ofrecen varias ventajas, como especialmente la implementación de ciertos requerimientos de forma mucho más simplificada que en la programación web tradicional. Sin embargo, esta característica, como es implementada en Links, implica el envío de toda la información de estado de la aplicación, en cada comunicación del cliente con el servidor. Además, esta información se almacena totalmente en el cliente. El servidor web de Links, no soporta Https, por lo tanto, toda información que se transmite entre cliente y servidor, es posible de ser escuchada por cualquiera que tenga acceso al canal de comunicación. Además, la información de continuación, no está cifrada originalmente por el servidor, sino solamente codificada en Base64 [6], de donde es muy fácilmente accesible, y modificable [27]. Toda la información de estado se almacena en un parámetro *_k*, presente en la URL de toda página o en un *input* tipo *hidden* del mismo nombre [27]. Por lo tanto leyendo este valor, como se indicó, es posible acceder información sensible, y además, modificándolo, es posible cambiar la forma de ejecución del servidor, al ejecutar un requerimiento del cliente.

Así por ejemplo métodos de modificación de la base de datos del servidor, que en la programación tradicional solo son ejecutados desde la capa de servicios, están expuestos totalmente a un cliente malicioso. Este podría ejecutar todas las acciones que se podrían ejecutar desde el propio programa Links.

Veamos algunos ataques que se pueden realizar usando un cliente malicioso [27]:

- Se puede obtener información secreta del programa en Links analizando la página web.
- Cambiando el valor de la continuación, se puede cambiar la información enviada al servidor, y ejecutar funciones con valores particulares.
- Se puede llamar a una función, no llamada originalmente al hacer submit

Estos ataques fueron tempranamente reportados en Links, 2009. Por lo que se implementó el primer ataque a modo de prueba actual de esta situación. Esta prueba es accesible en el ejemplo 9 del código web.links:

```
fun buy(value, dbpass) server {
  page
   <#>{intToXml(value)}</#>
}

fun sellAt(price) server {
  var dbpass = "secret";
  page
   <html>
    <form l:action="{buy(price,dbpass)}" method="POST">
      <button type="submit">Buy</button>
    </form>
```

```
    <div id="bar"/>
    <a href="src/web.links" target="_blank">código fuente, ver page9()</a>
   </body>
  </html>
}
```

Si se llama a *sellAt(20)*, se crea una página web, con un campo *input* tipo *hidden* que contiene la continuación de la aplicación. Si se decodifica *base64* este string, puede verse claramente la palabra *"secret"*, presente, o sea es posible acceder a una 'variable' secreta del programa.

Existen propuestas de seguridad en Links, una provista por este mismo trabajo. Este propone básicamente incorporar cifrado a la comunicación entre cliente y servidor. De todas maneras esta opción, *TinyLinks*, muy básica, no incorpora funciones de acceso a la base de datos, además de cambiar la sintaxis de Links quitando la transparencia de ejecución provista por la característica de programación distribuida, se deben usar sintaxis especial al invocar funciones remotas.
Otra propuesta [28, 29] involucra restringir el acceso a ciertas funciones, las de acceso a la base de datos. De esta manera ya no todas las funciones son accesibles directamente por el cliente. Esta opción no provee cifrado en la comunicación, de tal manera que siguen existiendo problemas como el expuesto en el ejemplo anteriormente.

Por lo tanto, ninguna de las dos opciones es adecuada, y además, ninguna de las dos se ha incorporado a la versión oficial de Links.

El problema de la seguridad en Links, puede verse como el principal motivo detrás de la ausencia en Links de funciones de entrada/salida al sistema de archivos. Si fuera posible, con estas falencias en seguridad presentes, sería posible por un intruso, leer, modificar y borrar cualquier información del sistema de archivos al cual el proceso del servidor web Links tiene acceso [7]. La misma conclusión se llega con respecto a usar cualquier programa nativo del sistema operativo.

Aquí podemos ver una muy importante carencia de Links en este aspecto. Si la aplicación tiene como requisito de cierto peso la seguridad, Links no será factible para su desarrollo.

5.18 Acceso a la Base de Datos

Esta es una de las características más destacables del lenguaje, y de mucho peso en su evaluación. Para satisfacer el ya nombrado *type mismatch problem*, se crearon traductores para el front-end, Html-Css, ya analizado, y para el back-end.
Para el back-end, Links utiliza un traductor que convierte código en Links a SQL. Esto le provee al lenguaje, la característica llamada "*language integrated query*", código escrito en el propio lenguaje, permite hacer consultas a la base de datos. Esta es una característica muy poderosa y avanzada del lenguaje, y una tendencia desde hace algunos años, como se puede ver mediante la aparición de lenguajes como LINQ [12], o Ferry [58].

Ventajas

La principal ventaja que aporta, es que al incorporar una característica de consultas al lenguaje, junto al tipado fuerte, cualquier tipo de error sintáctico o de tipos en la formación de la consulta se conoce en tiempo de compilación, por lo que acelera y en mucho la corrección de una consulta a la base de datos.
Esto puede compararse con JDBC en Java, donde no se conoce si la consulta está bien formada hasta ejecutar esa parte del programa que la corre. Esto incluso ocurre con consultas en otro meta lenguaje de SQL, como HQL [59], el lenguaje de consultas de Hibernate [60], un framework muy usado en la programación Java. En el mismo las consultas son Strings, y solo se comprueba su correctitud hasta que se ejecuta el programa. Puede verse que una técnica para tratar de solucionar este problema en Java es ejecutar la consulta directamente contra la base de datos, en un programa externo. Otra si se usa HQL es ejecutar en otro ambiente más, como http://hibernate.org/tools/. Esto dificulta mucho la integración de las consultas al lenguaje, ya que es necesario primero probarlas en un ambiente separado, para lo cual es necesario configurar esta otra herramienta lo que no es muchas veces sencillo, luego copiar la consulta y ejecutarla en ese ambiente. Cualquier corrección debe hacerse en ese nuevo ambiente y luego copiarla nuevamente en el ambiente de desarrollo original del lenguaje, en este caso Java. Además la configuración del ambiente donde se hacen estas pruebas, en caso de HQL, puede no ser exactamente el mismo que el ambiente donde está el programa, es necesario estar al día con todos lo mapeos de los objetos a la base de datos, y tener esos objetos Java actualizados. Esto es un inconveniente importante. Nada de esto ocurre con Links.

En Links, tan pronto se manda correr el programa, si la consulta a la base de datos está mal formada, salta un error de compilación. Esta es una ventaja de mucho peso, mejora tiempos de desarrollo, correctitud, legibilidad y mantenibilidad. Puede verse como fue nombrado incorporada a plataformas de lenguajes comerciales como .NET (en LINQ).

Otra ventaja de este enfoque, relacionado con el problema de *impedance mismatch*, es que alcanza con aprender un lenguaje, Links, no es necesario aprender SQL, con sus tipos y sintaxis. Si a SQL se le suman Javascript, vemos que la ventaja es importante.

También este enfoque permite incorporar soluciones más eficientes a problemas comunes en la programación. Por ejemplo es sabido que SQL también tiene sus limitaciones, no hay recursión, no permite listas anidadas, sino que el resultado debe ser todo plano, cada elemento de una fila debe ser de un tipo básico SQL. Por ejemplo si hay una relación entre departamentos y empleados, y se quiere una consulta que por departamento muestre todos sus empleados, en SQL se puede hacer un join de departamentos y empleados. De esta manera se repetirán departamentos, aparecerá tantas veces un departamento como empleados haya en ese departamento. Muchas veces a nivel de programación a alto nivel, lo que se quiere es un lista de departamentos, donde cada elemento que es un departamento, a su vez tenga una lista con sus empleados. Esta representación parece más natural a la hora de modelar el problema. Como mencionamos, esto no se puede hacer en SQL, y si el lenguaje del servidor es Java, se debe realizar una consulta SQL como se describió, cargar todo en memoria, recorrer el resultado agrupando por departamento y así

crear dinámicamente la nueva lista de listas deseada. Esto es muy desventajoso, es poco eficiente, es propenso a errores, ya que se puede cometer cualquier error al programar la conversión y es costoso para el programador, atenta contra la legibilidad y mantenibilidad, ya que tiene que realizar estas conversiones cada vez que requiera este tipo de modelado. Esto no ocurre en Links, Links además de nativamente proveer consultas para la base de datos, que se traducen en SQL, también incorporó recientemente esta característica [10]. Esto permite agregar a la solución al problema de traducir a SQL un lenguaje, la capacidad de consultas anidadas, un modelado que surge frecuentemente y no puede ser resuelto por SQL puro. Es una característica muy avanzada del lenguaje.

Otra ventaja es la reutilización, es posible incorporar a las consultas, funciones que se usen además para otra parte del programa [10].

Además el pasar parámetros a las consultas se hace a través del propio lenguaje, funciones que devuelven valores que se incorporan a las consultas. No se concatena simplemente un parámetro como un *String* al código SQL, como podría hacerse en Java o PHP. Esto evita problemas de seguridad como inyección de SQL.

Todas estas características proveen a Links de grandes ventajas, como mejora en tiempos de desarrollo, correctitud, legibilidad y mantenibilidad.

Limitaciones

Junto con las ventajas, también aparecen limitaciones en el lenguaje.

Analizemos algunos ejemplos:

```
fun getFactorialsTable()
{
  var db = database "factorials" "postgresql" ":5432:usuario:";
  table "factorials" with (i : Int, f : Int) from db
}
```

Esta es la manera de Links de acceder a una tabla de base de datos. Hay una función que devuelve la base de datos y otra que devuelve la tabla. Se observa que no está presente el manejo de posibles errores. El programador no puede indicar que hacer en caso de falla de autenticación, pérdida de conexión, timeout, etc. También este tipo de errores saltan a la hora de ejecutar esta porción de programa, no en tiempo de compilación.

Tampoco presenta aspectos más avanzados de programación web como caché de base de datos [61] que permiten mejorar performance, además de incluir *pool* de conexiones, lo que optimiza los recursos del sistema.

También se observa especialmente, que los mapeos de las tablas a los tipos en Links son solo a través de *records,* donde a su vez cada record tiene un tipo básico. En el anterior ejemplo se pueden observar como *(i : Int, f : Int).* Así, en caso de que el programador decidiera que el tipo de dato que va a manejar en el servidor es diferente de estos, como

por ejemplo si contiene enumerados, o sea variantes, ya no es posible realizarlo. Lo mismo ocurre si el tipo deseado contiene relaciones con otros tipos, lo que se correspondería mejor con una base de datos relacional. Esto no es posible realizarlo en Links. Igual situación ocurre si el tipo de datos no está presente en Links, pero sí en la base de datos. Esto ocurrió en la implementación de la historia agregar estudiante, donde era necesario incluir una fecha. La ausencia del tipo fecha en Links, obligó a mapear a *String* en Links este tipo *Date* en la base de datos.

Estas situaciones no ocurren en Java con Hibernate. Es posible mapear todos los tipos de la base de datos a los tipos en Java, incluso enumerados y tipos más complejos, y además y muy especialmente, es posible mapear las relaciones en la base de datos a relaciones entre los tipos. Así es posible dado un objeto, navegar hacia sus objetos relacionados a alto nivel, (en el lenguaje anfitrión, Java), sin necesidad para el programador de realizar consultas particulares a la base de datos con este propósito. Por ejemplo, si nuestro modelo consta de personas y autos, y es una relación uno a muchos, desde persona, es posible obtener una lista con sus autos, solo navegando hacia la lista desde el objeto persona. Lo mismo ocurre si desde el objeto auto, se quiere conocer su propietario. Basta navegar en su relación al objeto persona. Esto se realiza, al costo de haber previamente mapeado todas las tablas y relaciones en la base de datos a los objetos en el lenguaje anfitrión.

Así vemos como la solución planteada del *impedance mismatch problem*, es una solución parcial en este caso, en cuanto a evitar la conversión de tipos. No es posible realizar una total correspondencia de los tipos en SQL a los tipos en Links, primero porque Links no posee ciertos tipos, y segundo porque solo se pueden mapear tipos básicos. Y además, tampoco es posible mapear las relaciones de los tipos. De querer o necesitarse mapear a tipos más complejos, es necesario hacerlo manualmente. Esto en caso de un proyecto con muchas relaciones, y que requiera muchas de estas consultas, lleva a un aumento de los tiempos de desarrollo en Links. Por otro lado, si se requieren pocas consultas, y los tipos a convertir son los tipos básicos de Links, el tiempo de desarrollo sería menor en Links.

Consultas a la base de datos

Ahora veamos una consulta a la base de datos:

```
fun fact(n)
{
  var factorials = getFactorialsTable();
  var u = for (r <-- factorials)
              where (r.i == n)
             [r];
  hd(u).f
}
```

Aquí las observaciones en cuanto al mapeo de las tablas en la base de datos a los objetos en el lenguaje anfitrión, realizadas anteriormente son especialmente válidas, muchas consultas se pueden simplificar si se dispone de funcionalidades similares a Hibernate.

También como se ha nombrado anteriormente, se dispone de un lenguaje de consultas sobre los objetos mapeados, HQL.

Importa observar, que la programación web tradicional, no permite mediante la interfaz nombrada de Hibernate o HQL abstraer los suficiente para poder realizar cualquier consulta a la base de datos con esta interfaz. HQL es menos expresivo que SQL. O sea, ciertas consultas SQL muy elaboradas no pueden trasladarse al lenguaje de interfaz HQL (por ejemplo tablas temporales). En estos casos, a pesar de no ser muy frecuentes, se permite el uso directo de SQL contra la base de datos. También lo permite LINQ [12]. Links en cambio, no lo permite, no se puede usar SQL directo. Esto habla de cierta rigidez en el diseño, lo que simplemente vuelve no factible o potencialmente no factible al lenguaje en cuanto a requerimientos no previstos originalmente.

Volviendo al ejemplo *fact(n)* presentado más arriba, vemos que todo el *for* de Links es traducido a una consulta SQL de la base de datos. Es importante destacar que esta traducción tiene muy importantes limitaciones:

1. No es posible utilizar funciones nativas de la base de datos, como se ve más adelante en esta sección
2. No es posible utilizar funciones de agregación, como *sum*, *average*, etc. [6]
3. Si se incluyen funciones de Links que no existen en la base de datos, esta consulta no se puede traducir. Estas funciones son las que tienen efecto *wild,* especialmente creado con este propósito. Por ejemplo la función *explode*.
4. No se puede hacer *outer join.* No es posible realizar *left outer join*, algo que ocurre frecuentemente en requerimientos típicos.
5. No soporta valores nulos [11].
6. No hay funciones para crear tablas, modificar su estructura, borrarlas, etc. [9]
7. No se puede usar subconsultas. O sea que en el "*from*" traducido*,* aparezcan otras consultas. Siempre tienen que ser tablas, como se observa en la figura de la subgramática SQL más adelante.
8. No dispone de transacciones en la base de datos.
9. Solo se soporta oficialmente una BDs, Postgresql, ninguna otra [31].
10. *orderby* no soporta ordenar por más de un campo, y el orden es siempre ascendente.

O sea hay muy importantes limitaciones al hacer una consulta, simplemente algunas no es posible hacerlas, como crear una tabla, u ordenar en forma descendente.

En la primera etapa de pruebas sobre el lenguaje, se construyeron casos de prueba que muestran que existen funciones que no se traducen a consultas en la base de datos, este es un ejemplo:

```
# se puede llamar a un select en la BDs con funciones de la BDs?
fun dbSelectCond()
{
  var factorials = getFactorialsTable();
  for (r <-- factorials)
```

ni abs(intToFloat(r.i)) > 3.5 ni exp(intToFloat(r.i)) > 10. funcionan, no compila, así estas funciones no son trasladables a la BDs.

 # abs es de tipo int -> Int cuando en postgresql es de Float->Float y exp no existe directamente en Links.

where (abs(r.i) > 3)

[r]

}

Estas limitaciones observadas, la mayoría de ellas inferidas desde la propia sintaxis y características de Links más en algunos casos pruebas sobre el lenguaje, se pueden explicar de manera mucho más clara, observando algunos documentos que hablan directamente del subconjunto de SQL que puede generar Links [2]:

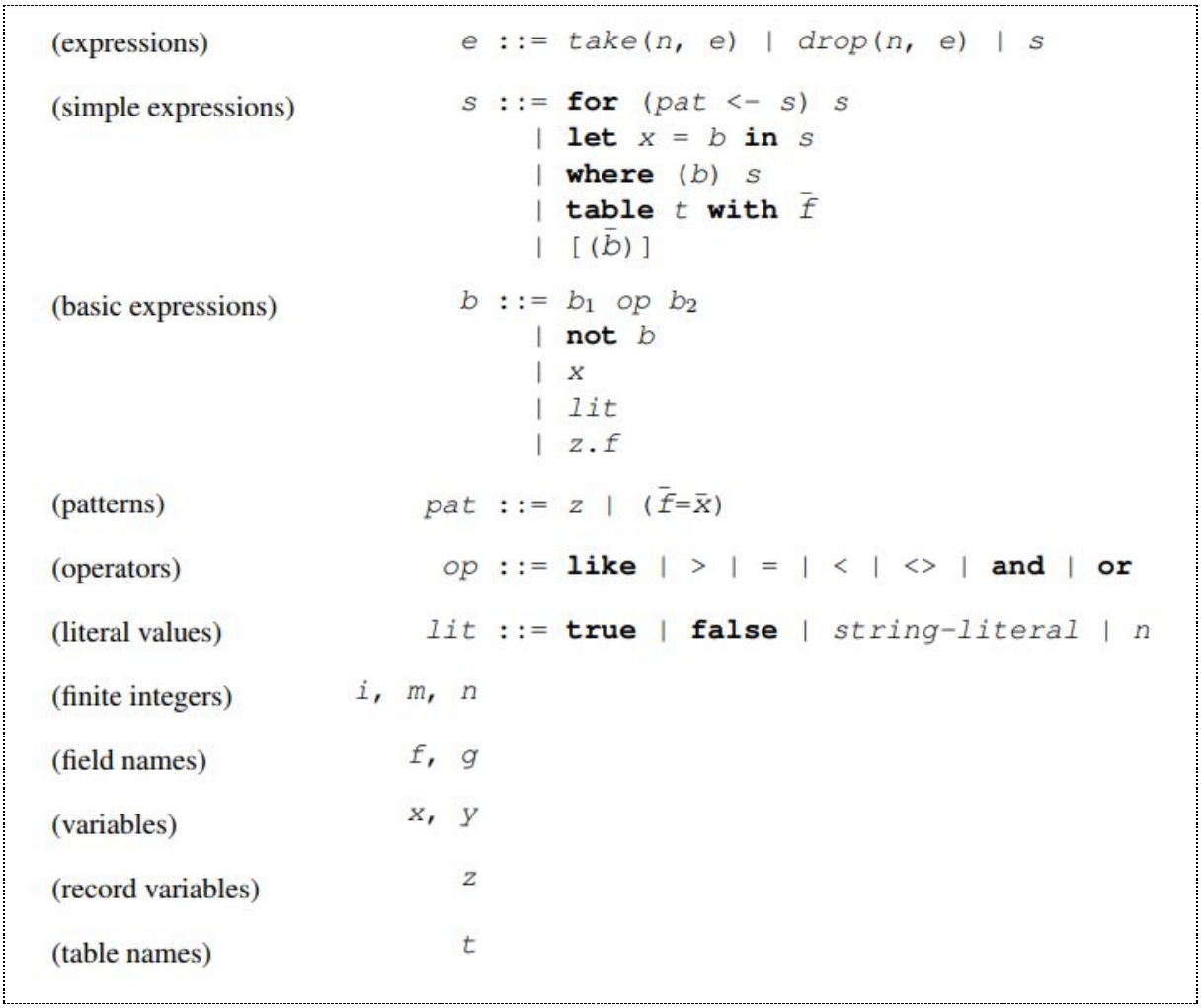

Figura 10, gramática del subconjunto de Links traducible a SQL [2].

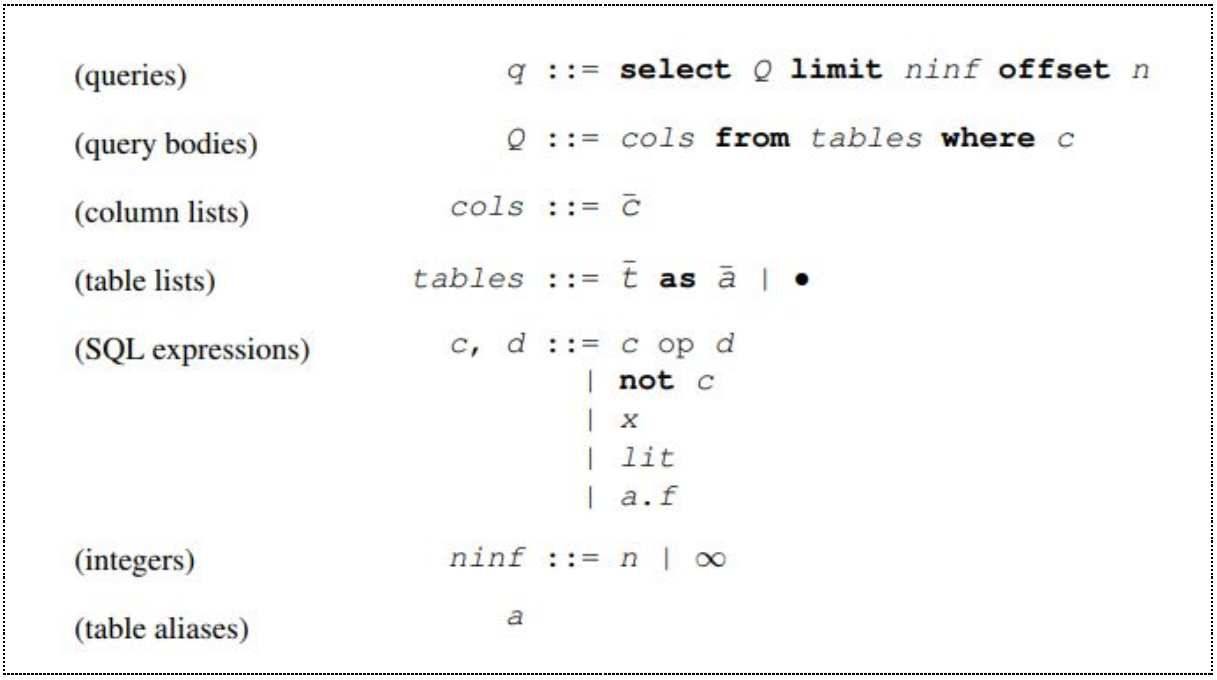

Figura 11, gramática SQL que resulta de la compilación de Links [2].

Puede verse que el subconjunto de SQL generado es limitado, por ejemplo se observa que directamente no es posible utilizar ninguna función de la base de datos. La única modificación que se hizo más adelante con respecto a la base de datos es *query shredding* [31], que utiliza más de una consulta. El subconjunto de Links que se traduce a una única consulta SQL sigue siendo el mismo.

En concreto en el caso de estudio, no fue posible usar las funciones de agregación para obtener el reporte de cantidad de cursos por estudiante. Se usó *query shredding*, que realiza dos consultas, no una, para obtener una lista de estudiantes donde cada fila a su vez contenía una lista de sus cursos. Luego con una función map se creó otra lista con los estudiantes y su cantidad de cursos (el largo de la sublista cursos). Esto en caso de ser muchos estudiantes y cursos, es altamente ineficiente, tuvo que traerse los estudiantes y cursos a memoria, y luego procesarlos.
También hubo similares situaciones con la historia de inscripción de estudiante a curso. En este caso es necesario crear una tabla con los cursos, e información de si el estudiante actualmente autenticado está inscripto o no en el mismo. Nuevamente se implementó con *query shredding.* Hubo que crear una lista de cursos con una sublista de las inscripciones del alumno. Luego con una función *map* convertir esta sublista al valor inscripto o no inscripto.

En destacable también hacer notar respecto a *query shredding*, la presencia de ciertos errores internos al lenguaje. Por ejemplo si se corre el archivo de pruebas unitarias *pruebas.links*, con la opción de configuración de *query shredding* activada (*shredding=on*) aparece este error:

...
Called from file "core/errors.pp.ml", line 114, characters 4-16
**** Error: "Assert_failure core/queryshredding.ml:1147:8"*

el cual no ocurre si no se usa esta opción. Esto habla de la cierta inestabilidad de esta característica del lenguaje. No se profundizó sobre este punto, al no considerarse crítico entrar en detalles al respecto.

Como se observa, se utilizó *query shredding* en nuestro caso, pero aún en otras situaciones más complejas, como cálculos con funciones, existe una alternativa. Implica recuperar toda la tabla, y luego aplicar todos los filtros, orden y otras funciones programáticamente con Links. Esta alternativa es muy poco viable en la industria, ya que es muy poco performante. Las bases de datos están creadas para optimizar este tipo de consultas, cuentan con índices por ejemplo, que permiten filtrados mucho más rápidos. También la representación de valores de Links presenta un importante *overhead* en memoria, especialmente para los valores resultado de consulta a la base de datos [22]. Por lo tanto, la opción de traer todo a memoria, implica el uso intensivo de este recurso, lo que no es viable en casos de tablas de gran volumen.

Aquí puede verse una limitación muy importante en Links. Se observa que la última alternativa en caso de no poder trasladarse una expresión *for* a la base de datos es comparativamente de muy baja performance, en cuanto a tiempo de procesamiento y utilización de memoria.

Modificaciones a la base de datos

Analizemos ahora las modificaciones en la base de datos:

```
fun dbInsert()
{
  var factorials = getFactorialsTable();
  insert factorials values (i,f) [(i = 30, f = 1)]
}

fun dbUpdate()
{
  var factorials = getFactorialsTable();
  update (r <-- factorials)
   where (r.i == 30 && r.f == 1)
   set (i = 30, f = -1)
}

fun dbDelete()
{
  var factorials = getFactorialsTable();
  delete (r <-- factorials)
   where (r.i == 30 && r.f == -1)
}
```

Las anteriores funciones son ejemplos de *insert*, *update* y *delete* en una base de datos. Aquí aplican las mismas observaciones hechas anteriormente para consultas a través de la expresión *for*.
La carencia de un ORM, hace comparativamente más compleja la persistencia. Por ejemplo si se quiere persistir una persona y sus autos, aquí hay que hacerlo con cierto orden, primero la persona y después cada uno de sus autos. Suponiendo un modelo en que los autos tienen una referencia a la persona, por la relación uno a muchos. Todo esto debe resolverse manualmente y además debe traducirse cada campo del tipo persona o auto a los campos en la base de datos. Por otro lado persistir esto usando Hibernate, consiste solo en persistir la persona, ya que los autos se pueden persistir por cascada. Y los mapeos objeto - tabla se hacen un única vez en un archivo de configuración.

También pueden aparecer problemas en caso de que en la base se usen *foreign keys* y referenciamiento mutuo. Si por ejemplo la entidad de la base de datos A hace referencia a la B y la B a la A, y se tiene que actualizar la base de datos, agregando las dos entidades A y B, no va a ser simple hacerlo. Para agregar A, que tiene una referencia a B obligatoria, tiene que estar ya agregada B. Y para agregar B, pasa lo mismo con respecto a A. En este caso, la manera usual de hacerlo, es deshabilitando temporalmente el chequeo de *foreign keys*. Luego hacer los cambios en la base de datos, y luego volver a habilitarlo. Esto suele hacerse en sistemas de respaldo y recuperación de la base de datos. En caso de Links, deberíamos hacer agregar A con una referencia a una entidad C previamente agregada, luego agregar B con la referencia a A, y luego modificar A actualizando su referencia a B.

Otro punto muy importante en caso de modificaciones es la interacción con las características ACID de la base de datos. En concreto, Links no posee transacciones sobre la base de datos. Vimos el ejemplo de la persona y sus autos. Tiene que realizarse primero el *insert* de la persona, y luego el de sus autos. Si es exitoso el de la persona, pero falla el de los autos, la base de datos queda en un estado inconsistente, queda una persona sin autos, cuando en realidad los tiene. Este problema en este caso puede no parecer tan crítico, pero en situaciones donde realmente sí lo es, puede tener consecuencias muy negativas. Por ejemplo una factura donde se persisten ciertos artículos y otros no.
En todos los casos donde se requiera el uso de transacciones, simplemente no va a ser posible utilizar Links, lo que implica carencia de factibilidad para estos requisitos.

Otro limitante, también muy relevante, es que Links solo soporta oficialmente una base de datos, Postgresql. Si simplemente un requerimiento es usar otra base de datos, o tiene que interactuar con un sistema legado que usa otra, como MySQL, Oracle, etc, podrían haber importantes inconvenientes al respecto. Se entra en una situación donde o directamente no es posible, como Oracle, o los riesgos involucrados de usar otra base de datos son muy altos, ya que es soportada pero no oficialmente, como MySQL, afectando la factibilidad del producto a desarrollar. No se sabe que va a funcionar, y que no.

Conclusiones

A pesar de las fuertes ventajas en cuanto a chequeo sintáctico, de tipos de una consulta y también *query shredding* que lleva mejoras en legibilidad, mantenibilidad, correctitud y tiempo de desarrollo, las limitaciones son muy relevantes.

Estas limitantes de Links en el acceso a la base de datos son de gran peso en la evaluación del lenguaje. En cuanto a la eficiencia de las consultas, simplemente no es posible obtener de forma eficiente cierta información de la base de datos. Si la aplicación contiene un volumen mediano de datos, y las consultas son elaboradas, la aplicación se volverá comparativamente lenta, o incluso simplemente fallará, si hay que traer toda la información a memoria y el servidor no cuenta con la suficiente. Esto hace al lenguaje no factible en estas situaciones.
La ausencia de manejo de transacciones, junto al manejo de errores en la base de datos, lleva a que en caso de error, difícilmente se pueda recuperar del mismo. En el caso de personas y autos, y falla de persistencia de autos, si el sistema pudiera recuperarse de este error, se podría intentar por lo menos borrar la persona ya persistida, para tratar de llevar al sistema a un estado consistente. Esto tampoco se puede hacer, ya que no se pueden capturar los errores de Links, simplemente los muestra en pantalla. Esto lleva a que en caso de actualizaciones de la base de datos, donde aparezcan relaciones entre dos tablas, no va a ser posible satisfacer un requerimiento de consistencia de la base de datos. Lo que hace también no factible al lenguaje a este respecto.
Por lo tanto, la factibilidad del lenguaje solo aplica, a productos muy sencillos, con actualizaciones sencillas a la base de datos, y volumen de datos relativamente pequeño, que no escale, y exclusivamente en Postgresql.

A conclusiones similares se llega en [6], donde además se implementan consultas más elaboradas.

6 Conclusiones

Previamente a las conclusiones generales, incluimos una observación final sobre otra característica del lenguaje, un lenguaje experimental.

6.1 Un lenguaje experimental

Nos parece muy relevante para las conclusiones finales este punto, por lo que lo incluimos previo a las mismas. ¿Links es actualmente un lenguaje experimental o un lenguaje orientado a la industria? Intentaremos responder esta pregunta.

Links nació como un lenguaje orientado a la industria, ambicioso en el sentido que pretendía ser el siguiente Python o Java [14], objetivo que se repite en las primeras conferencias sobre el lenguaje [62]. En el dominio de las aplicaciones web, el objetivo era resolver esos problemas que fueron tan destacados como la necesidad de aprender muchos lenguajes para poder desarrollar una sola aplicación web, y convertirse en el lenguaje líder en este dominio.
Se cumplió una primera etapa de desarrollo del lenguaje hasta 2009 [8], y últimamente se incorporaron manejadores de efectos y especialmente *session types*, [31].

Es destacable la forma en que se implementaron en el lenguaje funcionalidades como *session types*, con manejo de errores incluidos, pero no hay posibilidad de manejar cualquier otro tipo de error nativo del lenguaje, como errores en la base de datos, o cualquier error en la comunicación en la llamada a una función remota, como se observó en la sección manejo de errores. Esto ya da pie, a afirmar que actualmente Links es un lenguaje experimental, donde esta orientación sustituyó la línea original de orientación a la industria. No se modificaron implementaciones anteriores, para darle uniformidad al lenguaje, agregando manejo de excepciones nativo. Esto debería ser prioritario, dada su mucho mayor frecuencia en la programación web actual. Lo mismo puede decirse acerca de la carencia de sobrecarga de funciones, o polimorfismo ad-hoc, esta no fue implementada por considerarse un problema conocido en su momento [6], y fue planeada su implementación futura. Hasta la fecha no se implementó, pero sí el nombrado *session types*.

También las recientes publicaciones presentadas en la página web oficial, son trabajos de alto contenido teórico [24][25] que se alejan del objetivo inicial de hacer de Links un lenguaje para el programador web común, como fue por ejemplo el hacer la sintaxis de Links parecida a Javascript.

Finalmente, en trabajos recientes, puede verse esta situación explicitada, los mismos desarrolladores del lenguaje e investigadores llaman al lenguaje ‘experimental’ o ‘prototipo de investigación’ [21] [22]. Esto habla claramente de la situación actual del lenguaje. Actualmente Links es un lenguaje académico experimental.

6.2 Conclusiones

Pudo verse en las secciones del capítulo discusión, como al final de cada una, presentábamos conclusiones respecto al punto evaluado en la misma. Estas conclusiones generales deben sintetizar, y presentar un punto de vista global de todos los resultados obtenidos.

Se presentaron las grandes ventajas de Links, el tipado fuerte más el usar un único lenguaje, brinda fuertes ventajas en general, como mayor facilidad de aprendizaje del lenguaje, legibilidad, mantenibilidad, favorecer la correctitud de los programas en él escritos, además de mejorar los tiempos de desarrollo.
Otra característica fueron las continuaciones, novedosa para el programador web promedio que favorece ampliamente la solución de ciertos problemas.

Sin embargo, pudo verse como el *type mismatch problem* no fue totalmente resuelto. Fueron muchos los problemas involucrados. La conversión de tipos es parcial, y simplificada, la expresividad de Links tanto en Javascript como SQL es limitada. Estas situaciones hablan de la dificultades de implementar un lenguaje con la ambición original. La ambición junto con otras limitaciones, como el obviar en el diseño la seguridad [27], fueron determinantes.
La seguridad es crítica, junto a limitaciones en Ajax y performance en SQL. Solo los problemas en estas características son más que suficientes para afirmar que Links no es factible para un amplio conjunto de aplicaciones. La seguridad lo hace especialmente no adecuado para un muy amplio conjunto.
Otro aspecto es la carencia de interfaz con el exterior, desde *web services* hasta funciones de entrada/salida en el sistema de archivos u otras aplicaciones del sistema operativo. Esto involucra un riesgo muy alto para cualquier programador que quiera por primera vez desarrollar en el lenguaje, ¿cómo se soluciona un problema no previsto en el lenguaje?. No es posible. Hay una rigidez en el diseño, quizás alimentada por el objetivo inicial de un lenguaje que sustituya a todos.

Pudo observarse que el conjunto de aplicaciones donde Links es factible, es muy reducido. Tienen que ser simples, sencillas, en Postgresql, con consultas SQL simples, con bajo volumen de datos y que no escalen, sin Ajax, sin interacción con el exterior como *web services* y aplicaciones del sistema operativo, y que no requieran de transacciones. Además la seguridad no debe ser un requerimiento y la performance en tanto en el cliente como en el servidor tampoco.
A estas limitantes hay que sumarle la falta de documentación, lo que dificulta incluir funcionalidades no presentes en los ejemplos.

Finalmente, sumado a estas situaciones, se agrega la nueva orientación del lenguaje. No se buscó corregir ciertas situaciones críticas, sino agregar funcionalidad no tan relevante para el programador web promedio, como manejo de errores en *session types*. Así puede verse como actualmente es un lenguaje experimental, en general no adecuado para el desarrollo comercial.

Como puntos pendientes, se plantea la no investigación de animaciones en Javascript. Este punto se considera no central en la evaluación del lenguaje.

En cuanto a mejoras, dadas las grandes ventajas originalmente observadas, y que todavía existen los mismos problemas también originalmente observados en la programación web actual, llama la atención que no se hayan tratado de resolver algunos de los mismos en Links.
Se plantea por ejemplo la seguridad, los ejemplos de TinyLinks [27] más SELinks [28], parecen muy razonables de ser considerados como prototipos para incluir seguridad. Especialmente el uso de Https, más agregado de encriptación a la codificación de continuaciones, no parecen de gran dificultad implementativa, sin embargo, no se realizaron.

Como conclusión final, a pesar de las ventajas planteadas, se considera que la característica de factibilidad hace a Links no efectivo en la programación web actual. A pesar de ser muy prometedor, la dirección actualmente tomada por el lenguaje hace esperar un cambio en este aspecto, para poder cambiar esta conclusión.

7 Referencias bibliográficas

[1] Referencia del lenguaje Links.
http://links-lang.org/quick-help.html, 05/05/2018.

[2] Links: web programming without tiers
Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. FMCO'06 Proceedings of the 5th international conference on Formal methods for components and objects, Lecture Notes in Computer Science, vol. 4709, pages 266-296, 2007.

[3] Continuation in Racket (ex PTL Scheme)
http://docs.racket-lang.org/guide/conts.html?q=continuation#%28tech._continuation%29, 07/05/2018.

[4] Continuation in Racket 2 (ex PTL Scheme) 2
http://docs.racket-lang.org/more/index.html#%28part._.Continuations%29, 07/05/2018.

[5] Encuestas de desarrolladores 2017
https://insights.stackoverflow.com/survey/2017#most-popular-technologies, 05/06/2018

[6] Creating linksCollab: an assessment of Links as a web development language.
Steve Strugnell. 4th Year Project Report,Computer Science and Management Science, University of Edinburgh, 2008.

[7] LODE: an online IDE for Links in Links.
Carl Andersson. 4th Year Project Report, Artificial Intelligence and Computer Science, University of Edinburgh, 2008.

[8] Links sitio oficial. http://links-lang.org/, 06/07/2018

[9] DBWiki: a database wiki prototyped in Links. James Cheney, Sam Lindley and Heiko Müllr. 13th International Symposium on Database Programming Languages. Seattle, Washington, USA, 2011.

[10] Query shredding: efficient relational evaluation of queries over nested multisets. James Cheney, Sam Lindley, and Philip Wadler. ACM SIGMOD/PODS. Snowbird, Utah, USA, 2014.

[11] Row-based effect types for database integration. Sam Lindley and James Cheney. Seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation Philadelphia, PA, USA, 2012.

[12] LINQ native queries
https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/how-to-directly-execute-sql-queries, 06/06/2018

[13] Language-integrated provenance. Stefan Fehrenbach and James Cheney. 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, 2016.

[14] Original grant proposal for EPSRC. Philip Wadler, University of Edinburgh, 2005.

[15] The SLinks language. Gilles Dubochet, École Polytechnique Fédérale de Lausanne, 2005.

[16] Handlers for Algebraic Effects in Links. Daniel Hillerström, University of Edinburgh, 2015.

[17] Compilation of Effect Handlers and their Applications in Concurrency. Daniel Hillerström , University of Edinburgh, 2016.

[18] Compiling Links server-side code. Steven Holmes, University of Edinburgh, 2009.

[19] Row-based effect types for database integration. Sam Lindley and James Cheney. Seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation Philadelphia, PA, USA, 2012.

[20] Effective quotation. James Cheney, Sam Lindley, Gabriel Radanne, and Philip Wadler. Workshop on Partial Evaluation and Program Manipulation, San Diego, California, USA, 2014.

[21] Compiling Links Effect Handlers to the OCaml Backend. Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan. University of Edinburgh, 2016.

[22] Language-integrated provenance. Stefan Fehrenbach and James Cheney. 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, 2016.

[23] Objects and Aspects: Row Polymorphism Neel Krishnaswami, Department of Computer Science Carnegie Mellon University.
https://www.cs.cmu.edu/~neelk/rows.pdf, 06/06/2017.

[24] Lightweight functional session types. Sam Lindley and J. Garrett Morris, University of Edinburgh. In Behavioural Types: from Theory to Tools , River Publishers, 2017.

[25] Continuation Passing Style for Effect Handlers. Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Second International Conference on Formal Structures for Computation and Deduction, Oxford, United Kingdom, 2017.

[26] Session Types without Tiers Simon Fowler, Sam Lindley, J. Garrett Morris, Sára Decova University of Edinburgh, 2017.

[27] TinyLinks, Secure Compilation of a Multi-Tier Web Language. Ioannis G. Baltopoulos, Andrew D. Gordon, University of Cambridge, 2009.

[28] SELinks, Cross-tier, Label-based Security Enforcement for Web Applications. Brian J. Corcoran, Nikhil Swamy, Michael Hicks. University of Maryland, 2008.

[29] SELinks. http://www.cs.umd.edu/projects/PL/selinks/, 06/06/2018.

[30] Links Wiki GitHub. https://github.com/links-lang/links/wiki/, 06/07/2018.

[31] Links releases. https://github.com/links-lang/links/releases, 15/06/2018.

[32] PLT Scheme. http://plt-scheme.org/, 15/06/2018.

[33] A Basis for Concurrency and Distribution - From Data Types to Session Types http://groups.inf.ed.ac.uk/abcd/, 15/06/2018.

[34] Functional reactive animation in SVG for the web via Links . Chi-Feng Chou. University of Edinburgh, 2011.

[35] OPAM: OCaml Package Manager. https://opam.ocaml.org/, 26/06/2018.

[36] Apache Maven, https://maven.apache.org/, 15/06/2018.

[37] DOM. https://www.w3.org/2005/03/DOM3Core-es/introduccion.html, 15/06/2018.

[38] Haskell: The craft of functional programming. Simon Thompson. 2011

[39] Bootstrap 3. http://getbootstrap.com/docs/3.3/, 15/06/2018.

[40] Html 5. https://www.w3.org/TR/html50/forms.html, 15/06/2018.

[41] Apache Struts. https://struts.apache.org/, 15/06/2018.

[42] Apache Tiles. https://tiles.apache.org/, 15/06/2018.

[43] Node.js. https://nodejs.org/es/, 15/06/2018.

[44] Mustache.js. https://github.com/janl/mustache.js/, 15/06/2018.

[45] Less.js. http://lesscss.org/, 15/06/2018.

[46] Sass. https://sass-lang.com/install, 15/06/2018.

[47] Css 3. https://www.w3.org/TR/css3-roadmap/, 15/06/2018.

[48] Links Issues List. https://github.com/links-lang/links/issues, 15/06/2018.

[49] AngularJS. https://angularjs.org/, 15/06/2018.

[50] Elm. http://elm-lang.org, 15/06/2018.

[51] Java continuations - javaflow. http://commons.apache.org/sandbox/javaflow/, 15/06/2018.

[52] Compiling with Continuations. Andrew W. Appel. Princeton University, New Jersey, 2007.

[53] Spring. https://spring.io/, 15/06/2018.

[54] CORS. https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS, 15/06/2018.

[55] Web Storage. https://www.w3.org/TR/webstorage/#the-sessionstorage-attribute, 15/06/2018.

[56] Java Runtime Exceptions. https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/RuntimeException.html, 15/06/2018.

[57] Selenium. https://www.seleniumhq.org/, 15/06/2018.

[58] Ferry: Database-supported program execution. T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Universität Tübingen, Tübingen, Germany. In SIGMOD, June 2009.

[59] HQL http://docs.jboss.org/hibernate/orm/5.3/userguide/html_single/Hibernate_User_Guide.html#hql, 15/06/2018.

[60] Hibernate. http://hibernate.org/orm/, 15/06/2018.

[61] EhCache. http://www.ehcache.org/, 15/06/2018.

[62] Links, Web programming without tiers, talk slides. Philip Wadler. Melbourne, 2 Feb 2006.

Anexo A - Instalación del lenguaje Links

En este anexo describiremos los pasos necesarios para la instalación del lenguaje Links, en una máquina Amd64 con Debian 8, o 9, y en Raspberry Pi 3 con Debian 8.

Primeramente, es necesario instalar OPAM [35], una manejador de paquetes, que permite instalar OCaml, requerido para instalar a su vez Links v0.7.2.

Para instalar OPAM, una posibilidad, es usar apt-get, pero no es buena idea usarlo, ya que el paquete de instalación tiene la versión v1.2.0, que tiene sus problemas y es incompatible con la v1.2.2 que está muy estable [35], así que es preferible instalar la versión 1.2.2 y hacerlo directamente.

Es necesario instalar previamente a OPAM sus dependencias en el sistema operativo.

- Instalar dependencias de OPAM. Para eso sobre amd64 correr como root:

  apt-get install pkg-config libssl-dev libpq-dev git-core

  Además en Raspberry Pi 3 se necesita:

  apt-get install m4 aspcud

- Correr como usuario root:

  wget https://raw.github.com/ocaml/opam/master/shell/opam_installer.sh -O - | sh -s /usr/local/bin

  y:

  apt-get install make

- Correr como usuario común:

  `/usr/local/bin/opam init --comp 4.05.0`
  `eval `opam config env``

  `opam switch 4.04.0`
  `eval `opam config env``
  `opam config setup -a`

  Así quedó instalado OPAM 4.04 para el usuario local.

- Ahora se debe instalar PostgreSQL. Para Debian 8, correr como usuario root:

apt-get install postgresql-9.4 postgresql-client-9.4

En caso de usar Debian 9, correr:

apt-get install postgresql postgresql-client

- Se debe tratar de cerrar cualquier programa que consuma mucha memoria, como tomcat, java, etc previamente a correr lo siguiente. Correr como usuario común:

  opam install postgresql links-postgresql links

- Ahora ya corre sin la base de datos, con el comando:

  linx

  esto es un *prelude*, un ambiente interactivo, si se quiere correr el servidor, con ejemplos sin la base de datos, se puede correr:

  cd /home/<usuario>/.opam/4.04.0/share/links

  siendo <*usuario*> el usuario no root de instalación de OPAM.

  linx -m --path=examples:examples/games examples/webserver/examples-nodb.links

  el puerto 8080 tiene que estar libre.

- Para correr los ejemplos con la base de datos, se debe editar primero:

  /home/<usuario>/.opam/4.04.0/etc/links/config

  Y agregar las siguientes líneas:

  database_driver=postgresql
  database_args=:5432:usuario:

  donde *'usuario'* es el usuario actual no root.

- Cargar la base de datos. Copiar la carpeta de configuración de:

  /home/usuario/.opam/4.04.0/share/links/examples/dbsetup

  a:

  /tmp/dbsetup

con por ejemplo:

cp -r /home/usuario/.opam/4.04.0/share/links/examples/dbsetup /tmp/dbsetup

- Ahí correr con usuario *postgres* (su - postgres):

  ./createdbs
  ./populatedbs

  para crear las BDs y cargar los datos.

- Se debe agregar el usuario no root actual a postgres para poder usarlo en Links. Correr:

  su - postgres

  y:

  createuser --interactive

  o sea en lo anterior si el usuario actual es *'usuario'* en el interactivo a agregar el nombre tiene que ser *'usuario'*

- luego se debe crear bajo el usuario *postgres* una base de datos para el usuario *'usuario'*:

  createdb usuario

- Ahora bajo el usuario *‘usuario’* se pueden crear programas que conecten con la base de datos Postgres.

  Se pueden correr los ejemplos que usan la base de datos con:

  linx -m --path=examples:examples/games:examples/dictionary examples/webserver/examples.links

  En caso de no estar *linx* en el path, se puede agregar con:

  export PATH=$PATH:/home/<usuario>/.opam/4.04.0/bin

Recursos

http://opam.ocaml.org/doc/Install.html
https://opam.ocaml.org/doc/Usage.html
https://github.com/links-lang/links/blob/master/INSTALL
https://github.com/links-lang/links/wiki/Database-setup

Anexo B - Ambiente de desarrollo y producción

Links solo corre en Linux, con base de datos PostgreSQL, por lo que fue necesario instalar un ambiente bajo este sistema operativo y con esta base de datos.
Se instaló Debian 8, y luego de ciertos problemas con la máquina, Debian 9.

Para correr el servidor web, especialmente en producción, fue necesario crear un servicio en Debian, que se iniciara al iniciar la máquina.
También en producción fue necesario instalar un servicio de dns dinámico, y configurar routers.

Se usó como SCM Git, con Bitbucket como servidor remoto. Se investigaron varias interfaces gráficas para Git, como Git Extensions. Finalmente se optó por usarlo por línea de comandos.

IDE en Links

No existe un ambiente integrado de desarrollo en Links, el único que se creó [7], no está disponible, ni online, ni como código fuente. Por lo tanto, fue necesario investigar para poder lograr un ambiente de desarrollo aceptable.
Se probaron varios editores de texto, y finalmente se optó por Notepadqq. Es posible editar varios archivos a la vez, y es de fácil manejo.
Como era esperable, no disponía de resaltador de sintaxis para Links, pero se usó en su lugar el disponible para el lenguaje C, que dio buenos resultados.
La técnica para probar programas y ver resultados, fue modificar en Notepadqq, luego detener y correr el servidor Links mediante un terminal, y luego mirar los resultados en un navegador web, Google Chrome v67.