Links: Web Programming Without Tiers

Ezra Cooper   Sam Lindley   Philip Wadler   Jeremy Yallop
University of Edinburgh

Abstract

Links is a programming language for web applications. Links generates code for all three tiers of a web application from a single source, compiling into JavaScript to run on the client and into SQL to run on the database. Links provides support for rich clients running in what has been dubbed *'Ajax'* style. Links programs are *scalable* in the sense that session state is preserved in the client rather than the server, in contrast to other approaches such as Java Servlets or PLT Scheme.

1. Introduction

Ask a room of researchers whether they have purchased goods or services online. Every hand will go up. Now ask whether they have ever *failed* to complete some online purchase because of a fault in the web site software. Every hand will go up again.

Consumer spending on the web exceeded £12 billion per year in Britain, growing nearly 20 times faster than traditional retail [32]. Yet web programming remains an immature art, expensive and error prone. Even major sites like Orbitz, Apple, Continental, Hertz, and Microsoft suffer from fundamental problems [24].

This paper describes the design and implementation of Links, a programming language for web applications. A quarter century ago, Burstall, MacQueen, and Sannella at Edinburgh introduced an influential programming language, Hope [13]. Hope was named for Hope Park Square, located near the University on the Meadows. Links is named for the Bruntsfield Links, located at the other end of the the Meadows and site of the world's first public golf course.

A typical web system is organized in three tiers, each running on a separate computer (see Figure 1). Logic on the middle-tier server generates forms to send to a front-end browser and queries to send to a back-end database. The programmer must master a myriad of languages: the logic is written in a mixture of Java, Perl, PHP, and Python; the forms in HTML, XML, and JavaScript; and the queries are written in SQL or XQuery. There is no easy way to link these — to be sure that a form in HTML or a query in SQL produces data of a type that the logic in Java expects. This is called the *impedance mismatch* problem.

Links eliminates impedance mismatch by providing a single language for all three tiers. In the current version, Links translates into JavaScript to run on the browser and SQL to run on the database. The server component is written in O'Caml; it consists of a static analysis phase (including Hindley-Milner typechecking), a translator to JavaScript and SQL, and an interpreter for the Links code that remains on the server.

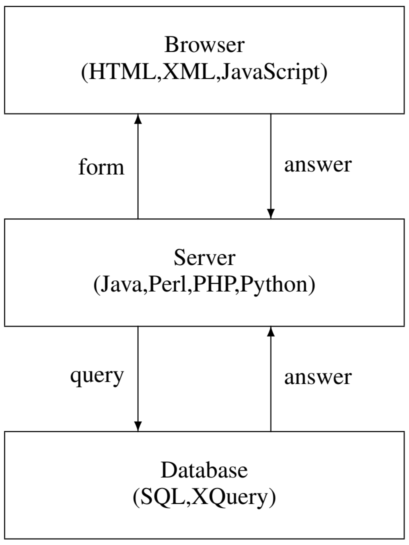


Figure 1. Three-tier model

Links is a strict functional language. Side effects play a limited but important role: they are used for updating the display, communication between concurrent processes, and updates to the database.

Increasingly, web applications designers are migrating work into the browser. "Rich client" systems include Google Mail, Google Maps, iRows, , using a new style of interaction recently dubbed "*Ajax*" [21].

Links currently compiles into JavaScript, a functional language widely available on most browsers. JavaScript is notoriously variable across platforms, so we have designed the compiler to target a common subset that is widely available. It would be easy to extend Links to support additional target languages, such as Flash or Java. However, we expect the popularity of Ajax will mean that more standard and reliable versions of JavaScript will become available over the next few years.

Links incorporates ideas proven in other functional languages, including support for:

- database programming, as found in Kleisli and elsewhere,
- web interaction, as found in PLT Scheme and elsewhere,
- concurrency, as found in Erlang and elsewhere, and
- XML programming, as found in XDuce and elsewhere.

A detailed description of these influences appears below.

Links also supports a number of features not found elsewhere.

- As mentioned above, the defining feature of Links is that one writes a program in a single language to run on all three tiers. Links code is translated to JavaScript to run on the client, and to SQL to run on the database.
- Links provides support for rich clients running in what has been dubbed *'Ajax'* style. Below we give several examples of interactive applications to demonstrate what can be achieved in Links: disabling the submit button unless all input data is valid; querying a database on every keystroke and displaying the first ten answers that match; draggable lists that can be rearranged by the user; and displaying on the client progress of a computation in the server. (See Section 2.)
- Links programs are *scalable* in the sense that session state is preserved in the client rather than the server. This contrasts with approaches such as Java Servlets, which keep a session thread running on the server, or PLT Scheme, which must store a continuation on the server. In Links, all server state is serialized and passed to the client, then restored to the server when required. This *resumption passing style* extends the *continuation passing style* commonly used in PLT Scheme and elsewhere. Links functions are labelled as to whether they are intended to execute on the client or the server; functions running on the client may invoke those on the server, and vice-versa. (See Section 3.1.)
- Links uses techniques pioneered in Kleisli to express queries in a functional language and compile these into SQL. As well as demonstrating that these techniques can be applied to web applications (Kleisli was applied to bioinformatics), Links also includes a number of improvements over the technique used in Kleisli. In particular, for display of data on the web it is often important to be able to quickly select a portion of the data returned by a query. Links achieves this by compiling `take` and `drop` operations on lists into features found in most implementations of SQL. (See Section 4.)

Links is a typed, functional language. Query optimization in Kleisli, pattern matching in XDuce, continuation-based web interfaces in Scheme, and distribution in Erlang all work better with immutable values rather than with mutable objects. Types ensure consistency between forms in the browser, logic in the server, and queries on the database; and between the sender and receiver of a message in a concurrent program.

Database programming Queries are written in the Links notation and compiled into SQL, a technique pioneered by Kleisli. Kleisli was developed by Buneman, Wong, and others, based on comprehensions for database query languages [12, 34, 60], which in turn were inspired by monad comprehensions [56]. Kleisli has been used to solve database integration problems that otherwise have proven intractable, has been applied to a range of problems in bioinformatics, and is sold as a commercial product. Other database systems that apply comprehensions for querying include Erlang's Mnesia [59] and work by Grust [28]. Recently, the LINQ framework for .NET was announced, which provides support similar to that offered in Links [35]. The similarity in names is a coincidence, but the similarity in techniques is not, as Links and LINQ both derive from roots in Haskell and other functional languages.

Web interaction The notion that a programming language could provide support for web interaction first appears in the programming language MAWL [2]. The notion of *continuation* from functional programming has been particularly fruitful, being applied by a number of researchers to improve interaction with a web client, including Quiennec [47, 48], Graham [23] (in a commercial system sold to Yahoo and widely used for building web stores), Felleisen and others [24, 25, 26], and Thiemann [54].

Links supports web interaction using *continuations* to capture the state of the interaction, and *types* to ensure consistency between the client and the server. Felleisen and others [24] have proposed a methodology for developing web applications in Java, based on *continuation passing style* and *defunctionalization*, and we apply similar techniques to compile Links programs. This results in *scalable* programs that maintain session state in the client rather than the server. Only a few commercial web tools (such as ASP.NET) produce code that is scalable in this sense; many commercial web tools (like J2EE) and most research web tools (including current releases of PLT Scheme [26] and Mozart QHTML [17]) are not scalable in this sense.

Concurrency Links supports concurrent programming in the client, using "share nothing" concurrency where the only way processes can exchange data is by message passing, as pioneered in Erlang. Concurrency in the client is implemented in a lightweight style, compiling into continuation-passing style and switching processes after a given number of function calls, rather than by using operating system threads. We intend to also support concurrency and message passing in the server, but this is an area for future work.

The functional language Erlang [1, 3] is used in Ericsson's AXD301 ATM phone switch. The AXD301 has 99.9999999% (nine 9s) reliability, making it one of the most reliable switches ever made, and it contains 1.7 million lines of Erlang, making it the largest functional program ever written. Erlang is used in a number of other market leading products, including Ericsson's GPRS and Alteon's Accelerator. The key to Erlang's success is its use of lightweight concurrency, message passing, and "share nothing" concurrency. Another language that supports lightweight processes and "share nothing" concurrency is Mozart [17, 55].

XML programming Links provides convenient syntax for constructing XML data, similar to that provided by XQuery [61]. The current version does not support regular expression types, as found in XDuce and other languages, but we expect to add them in a future version. Regular expression types were given a low priority, because they are already well understood from previous research.

XDuce was developed by Pierce, Hosoya, and others, based on regular expressions types for XML [30, 31]. XML types were further developed by Wadler and others in XQuery [18, 52, 61], by Schwartzbach and others in Bigwig and JWIG [6, 11, 14, 50], by Castagna and others in CDuce [9], and by Pierce and others in Xtatic [27]; a survey of type systems for XML was published by Møller and Schwartbach [39].

Other languages Other languages for web programming include Xtatic [27], Scala [43, 44], Mozart [17, 55], SML.NET [8], F♯ [53], and Cω (based on Polyphonic C♯ [7] and Xen [10]). These languages have many overlaps with Links, as they are also inspired by the functional programming community.

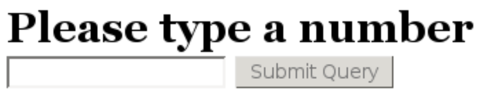
However, none of these languages shares Links' objective of generating code for all three tiers of a web application from a single source — scripts for the front-end client, logic for the middle-tier server, and queries for the back-end database. We expect that providing a single, unified language to replace the current multiplicity of languages will be a principal attraction of Links.

2. Links by example

This section introduces Links by a series of examples. The reader is encouraged to try these examples online at

`groups.inf.ed.ac.uk/links/examples/`

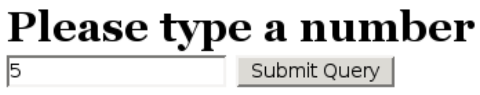
We begin with a simple factorial example to introduce the basic functionality of Links, and then present three further examples



Please type a number

Submit Query

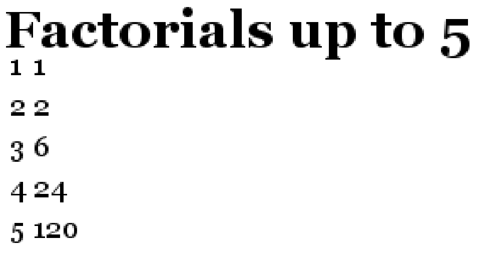
Figure 2. Form with submit button disabled



Please type a number

5 Submit Query

Figure 3. Form with submit button enabled



Factorials up to 5

```
1 1
2 2
3 6
4 24
5 120
```

Figure 4. Table of factorials

that demonstrate additional capabilities: a dictionary suggest application, a draggable list, and a progress bar.

2.1 Factorial

We begin with the "Hello World" of web applications: a program to display a table of factorials. The application presents a form with a single text field and asks the user to type a number. The submit button is disabled unless a valid number has been typed (Figures 2 and 3). On pressing submit, a table of factorials up to the given number is displayed (Figure 4). We have designed the program so that all computation of HTML and validation of the data takes place on the client. There is also a small twist to make the example more typical of a web application: rather than compute the factorials we find them by querying a database.

The Links code for this application appears in Figure 5. The application consists of three functions: `request`, to display the form, `response`, to display the table, and `lookup`, to compute the factorials. Where a given function is intended to execute is indicated by including the keywords **`client`** or **`server`** as appropriate in its header. Execution begins by evaluating the expression at the end of the code, in this case an invocation of the `request` function.

Implementation Links is implemented as an O'Caml program that runs as a single executable under a web server. The Links processor has five phases: parsing, type reconstruction, SQL generation, JavaScript generation and evaluation. We refer to the component that implements the evaluation phrase "interpreter" in this paper, and to the components that generate JavaScript and SQL as the "compiler".

Syntax The syntax of Links resembles that of JavaScript. This decision was made not because we are particularly fond of this syntax, but because we believe it will be familiar to our target audience. Functional programming is already sufficiently alien, without also requiring our users to read and write `f x y` rather than `f(x, y)`. Curried functions are an acquired taste.

One difference from the usual JavaScript syntax is that we do not use the keyword **`return`**, which is too heavy to support

```
table factorials (
 i:Int,
 f:Int
) database
  "postgresql:factorials::5432:www-data:";

fun request(s) client {
 domReplaceDocument(
  <html>
   <body>
    <h1>Please type a number</h1>
     <form l:onsubmit="{response(t)}"
      l:onkeyup="{request(t)}">
      <input type="text" value="{s}" l:name="{t}"/>
      {
      if (t ~ /[0-9]+/)
       <input type="submit"/>
      else
       <input type="submit" disabled="disabled"/>
      }
     </form>
    </body>
  </html>
 )
}

fun response(s) client {
 var n = stringToInt(s);
 domReplaceDocument(
  <html>
   <body>
    <h1>Factorials up to {intToXml(n)}</h1>
    <table>{
     for (var (i,f) <- lookup(n))
      <tr>
       <td>{intToXml(i)}</td>
       <td>{intToXml(f)}</td>
      </tr>
    }</table>
   </body>
  </html>
 )
}

fun lookup(n) server {
 for (var row <- factorials)
 where (row.i <= n)
 orderby (row.i)
  [(row.i, row.f)]
}

request("")
```

Figure 5. The factorial program in Links

a functional style. Instead, we indicate return values subtly, by omitting the trailing semicolon. The type checker indicates an error if a semicolon appears after an expression that returns any value other than the unit value `()`.

Links includes special syntax for constructing and manipulating XML data. XML data is written in ordinary XML notation, using curly braces to indicate embedded code. Embedded code may occur either in attributes or in the body of an element. The Links notation is similar to that used in XQuery, and has similar advantages. In particular, it is easy to paste XML boilerplate into Links code. The parser begins parsing XML when a < is immediately followed by a legal tag name; a space must always follow < when it is used as a comparison operator; legal XML does not permit a space after the < that opens a tag. Links also includes operators to find all children of a given element, or to find the element with a given `id` attribute. (Eventually, these operations will be written in XPath notation, but in the current implementation they are provided as library functions.)

DOM The client maintains an XML tree representing the current document to display; usually this will be in XHTML, the dialect of XML corresponding to HTML. This distinguished tree is referred to as the Document Object Model, or DOM for short.

Interaction The function *`request(s)`* displays the form, where the current value of the input is the string *`s`*. The submit button on the form will be enabled if the string *`s`* consists of one or more digits, as determined by matching against the regular expression `/[0-9]+/`, and disabled otherwise.

The following specifies the text input field on the form.

```
<input type="text" value="{s}" l:name="{t}"/>
```

The `value` attribute specifies that the initial value of the field is the string *`s`*, the `l:name` attribute specifies that the string typed into the field should be bound to a string variable *`t`*.

Why is the attributes `l:name` prefixed by `l:`, but the attribute `value` is not? The `value` attribute is computed by ordinary interpolation. The Links expression *`s`* is evaluated, and the result is taken as the value of the attribute. In contrast, for the `l:name` attribute the code between curly braces is not evaluated at all. Instead between the braces must lie a variable name, and that variable is bound to the contents of the field. Similarly, for the `l:onsubmit` and `l:onkeyup` attributes in the `form` tag, the code between the curly braces is not evaluated until the specified event occurs.

The attributes that Links treats specially are `l:name` and all the attributes connected with events: `l:onchange`, `l:onsubmit`, `l:onkeyup`, `l:onmousemove`, and so on. These attributes are prefixed with `l:`, using the usual XML notation for namespaces; in effect, Links presumes that `l` denotes a special Links namespace.

The scope of variables bound by `l:name` is the Links code that appears in the attributes connected with events. In the example, this means that the variable *`t`* is in scope for the code that appears in the `l:onkeyup` and `l:onsubmit` attributes.

The variables *`s`* and *`t`* denote successive values of the text field. The formal parameter *`s`* of *`request`* is used to initialize the text field. When the user changes focus after typing into this field, the variable *`t`* in the `l:name` attribute is bound to the new contents of the field, and the code in the `l:onchange` attribute is invoked, passing *`t`* as the actual parameter to the next call of *`request`*, binding *`s`* to *`t`* for the next call.

Links static checking ensures that a static error is raised if a name is used outside of its scope; this guarantees that the names mentioned on the form connect properly to the names referred to by the application logic. In this, Links resembles MAWL and Jwig, and differs from PLT Scheme or PHP. This style of interaction does not necessarily scale well, and it may be preferable to use a library of higher-order forms as developed in WASH or iData.

Display The Links code *`response(t)`* is invoked when the submit button is pressed. This in turn invokes *`lookup(n)`* to look up the values of the factorial function, and then formats an HTML table to display the results.

Both formatting and lookup are written using **`for`** loops, in the latter case also involving a **`where`** clause. These constructs correspond to what is written as a list comprehension in languages such as Haskell or Python. In general, if *`e`* evaluates to the list `[`*`v1`*`,...,`*`vn`*`]`, then the construct

```
for (var x <- e) f
```

evaluates to the list

```
f[v1/x] ++ ... ++ f[vn/x]
```

where *`f`*`[`*`v`*`/`*`x`*`]` substitutes value *`v`* for variable *`x`* in expression *`f`*, and ++ is list concatenation. A clause of the form

```
where (p) e
```

is exactly equivalent to

```
if (p) e
else []
```

A succession of **`for`** and **`where`** clauses may also have an **`orderby`** clause at the end. The code

```
for (var x <- e1)
orderby (e2)
 e3
```

is equivalent to

```
sortby(
 fun (x) {e3},
 for (var x <- e1) e2
)
```

where *`sortby`* is a library function[1] that applies the given function to each element of the list and then sorts on the given result. The **`orderby`** notation is conceptually easier for the user (since there is no need to repeat the bound variables) and technically easier for the compiler (because it is closer to the SQL notation that we compile into, as discussed in the next section).

Database In Links one may declare a table corresponding to a database. In this example, there is a table of factorials, where each row contains two integer fields, *`i`* and *`f`*, containing a number and its factorial, respectively. Within Links, each table is viewed as a list of records, and it may be iterated over using **`for`** and **`where`** clauses as one might expect. However, the Links compiler is designed to convert these into appropriate SQL queries over the relevant tables whenever possible. For example, the expression

```
for (var row <- factorials)
where (row.i <= n)
 [row]
```

compiles into the equivalent SQL statement

```
select row.i, row.f
from factorials as row
where row.i <= n.
```

This form of compilation was pioneered in systems such as Kleisli, and is also central to Microsoft's new Language Integrated Query (LINQ) for .NET. We discuss this compilation in detail in Section 4.

Client-server The keyword **`server`** in the definition of *`lookup`* causes invocations of that function to be evaluated on the server rather than the client. The function must be invoked on the server because we presume (as is often the case) that the database cannot be accessed directly from the client.

When this application is invoked, because the *`request`* function is marked as **`client`**, the server does nothing except to transmit the Links code (compiled into JavaScript) to the client. The client runs autonomously until the *`lookup`* function is invoked,

[1] In fact, the *sortby* library function itself is not implemented in the current version of Links.

at which point the `XMLHttpRequest` function is used to transmit the arguments to the server and creates a new process on the server to evaluate the call. No server resources are required until the *`lookup`* function is called. This contrasts with Java Servlets, which keep a process running on the server at all times, or with PLT Scheme, which keeps a continuation cached on the server.

Typing Links uses the Hindley-Milner type system [36, 16]. The variables *`s`* and *`t`* have inferred type *`String`*, the variable *`n`* has inferred type *`Int`*. Polymorphic types are supported (but don't play a significant role in this example).

Links currently gives all XML trees the same type, *`XML`*, which is equivalent to a list of XML items. XML types can be manipulated with the usual list operators, such as list append. We expect to support regular expression types for XML and strings in a future version of Links. Regular expression typing was not our first priority for work, since it is already well understood from other research efforts [31, 27, 9, 33].

Record and variant types with row polymorphism are also supported [46]. Rows of SQL tables are declared in Links with a corresponding record type, as in the line

```
(i:Int, f:Int)
```

declaring the table *`factorials`*.

The current version of Links also does not support any form of overloading. For this reason, the code requires explicit coercion functions such as *`stringToInt`* and *`intToXML`* (which is the composition of *`intToString`* and *`stringToXml`*). We expect to support overloading in future using a simplified form of type classes. Again, this was left to the future because it is well understood from other research efforts. In particular, the WASH and iData systems make effective use of type classes to support generic libraries [54, 45].

This completes our introduction to the basic features of Links. We now go on to see further examples of what can be achieved with this approach.

2.2 Dictionary suggest

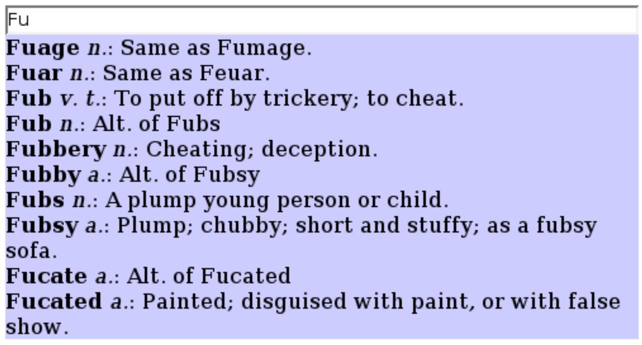
The dictionary suggest application presents a text box, into which the user may type a word. As the user types, the application displays a list of words that could complete the one being typed. A dictionary database is searched, and the first ten words beginning with the same prefix as the typed word are presented (see Figure 6).

This application is of interest because it must perform a database lookup and update the display at every keystroke. Applications such as Google Suggest have a similar structure.

The Links version is based on an ASP.NET version of the same application, available from [40], using the same dictionary and formatting. The dictionary contains 99,320 entries. Links is acceptably fast for this application, and response times for the Links and ASP.NET versions appear identical. (However, we have only tested with a lightly loaded server.)

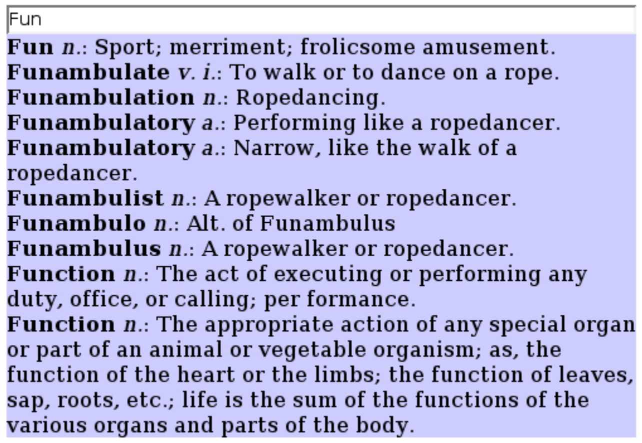
The code for the application is shown in Figure 8. The structure of the code is straightforward. On each keystroke, the function *`suggest`* is passed the current content of the text field. This function calls *`completions`* on the server to find the first ten words with the given prefix, and *`format`* on the client to format the list returned.

In contrast to the factorial application, this application is designed so that each keystroke triggers an action that replaces only the children of a named element of the document, not the whole document. This is done mainly for ease of expression, not for efficiency. In this case, the whole document involves only a few extra elements, and so replacing it would work just as well.


Fu

Fuage *n.*: Same as Fumage.
Fuar *n.*: Same as Feuar.
Fub *v. t.*: To put off by trickery; to cheat.
Fub *n.*: Alt. of Fubs
Fubbery *n.*: Cheating; deception.
Fubby *a.*: Alt. of Fubsy
Fubs *n.*: A plump young person or child.
Fubsy *a.*: Plump; chubby; short and stuffy; as a fubsy sofa.
Fucate *a.*: Alt. of Fucated
Fucated *a.*: Painted; disguised with paint, or with false show.

Figure 6. Dictionary suggest


Fun

Fun *n.*: Sport; merriment; frolicsome amusement.
Funambulate *v. i.*: To walk or to dance on a rope.
Funambulation *n.*: Ropedancing.
Funambulatory *a.*: Performing like a ropedancer.
Funambulatory *a.*: Narrow, like the walk of a ropedancer.
Funambulist *n.*: A ropewalker or ropedancer.
Funambulo *n.*: Alt. of Funambulus
Funambulus *n.*: A ropewalker or ropedancer.
Function *n.*: The act of executing or performing any duty, office, or calling; per formance.
Function *n.*: The appropriate action of any special organ or part of an animal or vegetable organism; as, the function of the heart or the limbs; the function of leaves, sap, roots, etc.; life is the sum of the functions of the various organs and parts of the body.

Figure 7. Dictionary suggest

Finding all words with a given prefix is indicated with the regular expression `/{prefix}.*/`. Curly braces are used for interpolation, as with XML; curly braces may also be used for interpolation into ordinary strings. In this case, the regular expression matches any string beginning with the given prefix followed by zero or more other characters (indicated by `.*`). This particular regular expression can be compiled into an instance of the `LIKE` operator provided by SQL, so that *`w.word ~ /{prefix}.*/`* becomes *`w.word LIKE {prefix}%`*.

2.3 Draggable list

The draggable list application displays an itemized list on the screen. The user may click on any item in the list and drag it to another location in the list (see Figures 9 and 10).

The code for draggable list is shown in Figure 11. The example exploits Links' ability to run concurrent processes in the client. Each draggable list is monitored by a separate process.

Each significant event on an item in the list (mouse up, mouse down, mouse out) has attached to it code that sends a message to the process indicating the event. The process itself is represented by two mutually recursive functions, corresponding to two states. The process starts in the waiting state; when the mouse is depressed it changes to the dragging state; and when the mouse is released it reverts to the waiting state. When the mouse is moved over an adjacent item while in the dragging state, the dragged item and the adjacent item are swapped.

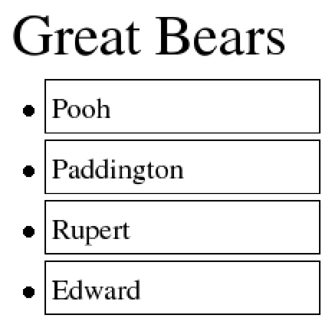
The function corresponding to the waiting state takes as a parameter the `id` of the element containing the draggable list; the

```
table wordlist : (
 word : String,
 type : String,
 meaning : String
) database
  "postgresql:dictionary::5432:www-data:";

fun suggest(prefix) client {
 domReplaceChildren(
  format(completions(prefix)),
  domGetRefById("suggestions")
 )
}

fun format(words) client {
 for (var w <- words)
  <span>
   <b>{stringToXml(w.word)}</b>
   <i>{stringToXml(w.type)}</i>:
   stringToXml(w.meaning)
   <br/>
  </span>
}

fun completions(prefix) server {
  where (prefix != "")
  take(10,
   for (var w <- wordlist)
   where (w.word ~ /{prefix}.*/)
   orderby (w.word)
    [w]
  )
}

<html>
 <head>
  <title>Dictionary suggest</title>
  <style>
   {stringToXml{"
   .input { width:500px }
   .suggestions { align:left; width:500px;
                  background-color:#ccccff }
   "}}
  </style>
 </head>
 <body>
  <h1>Dictionary suggest</h1>
  <form l:onkeyup="{suggest(prefix)}">
   <input type="text" l:name="{prefix}"
    class="input" autocomplete="off"/>
  </form>
  <div id="suggestions" class="suggestions"/>
 </body>
</html>
```

Figure 8. Dictionary suggest in Links

function corresponding to the dragging state takes the same parameter, and a second parameter indicating the item currently being dragged. Both functions are written in tail recursive style, each either calling itself (to remain in that state) or the other (to change state). This style of coding state for a process was taken from Erlang.

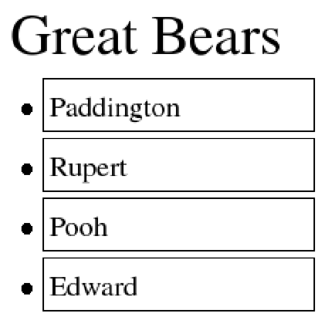
In this simple application, the state of the list can be recovered just by examining the text items in the list. In a more sophisticated application, one might wish to add another parameter representing the abstract contents of the list, which might be distinct from the items that appear in the display.



Great Bears

- Pooh
- Paddington
- Rupert
- Edward

Figure 9. Draggable list: before dragging



Great Bears

- Paddington
- Rupert
- Pooh
- Edward

Figure 10. Draggable list: after dragging

Observe that the code has been written so that there can be multiple draggable lists, each monitored by its own process.

2.4 Progress bar

The progress bar demonstrates an application where computation bounces back and forth between client and server. A computation is performed on the server, and the progress of this computation is demonstrated with a progress bar (see Figure 12). When the computation is completed, the answer is displayed (see Figure 13).

The code for the progress bar application is shown in Figure 14. In this case, the computation performed on the server is uninteresting, simply counting up to the number typed by the user. In a real application the actual computation chosen might be more interesting.

Periodically, the function `computation` (running on the server) invokes the function `showProgress` (running on the client) to update the display to indicate progress. When this happens, the state of the computation on the server is pickled and transmitted to the client. This is done by applying continuation passing style and defunctionalization to the code on the server; the client is passed the name of a function corresponding to the continuation of the computation, and the data needed to continue the computation. Note that no state whatsoever is maintained on the server when computation is moved to the client. The implementation is discussed in more detail in Section 3.1.

One advantage of writing the computation in this way is that if the client quits at some point during the computation (either by surfing to another page, terminating the browser, or taking a hammer to the hardware) then no further work will be required of the server.

On the other hand, the middle of an intensive computation may not be the best time to pickle the computation and ship it elsewhere. A more sophisticated design would asynchronously notify

```
fun waiting(id) {
 receive {
  case MouseUp -> waiting(id);
  case MouseDown(item) -> dragging(id,item);
  case MouseOver(newItem) -> waiting(id);
 }
}

fun dragging(id,item) {
 receive {
  case MouseUp -> waiting(id);
  case MouseDown(item) -> dragging(id,item);
  case MouseOut(newItem) -> {
   if (not(domIsNullRef(newItem)) &&
    domGetAttributeFromRef
     (domGetParentFromRef(newItem), "id") == id) {
    domSwapNodeRefs(newItem, item);
    dragging(id,item);
   }
   else waiting(id)
  };
 }
}

fun draggableList (id, items) client {
 var dragger = spawn { waiting(id) };
 <ul id="{id}"
  l:onmousedown="{
   dragger!MouseDown(event.target))}"
  l:onmouseup="{
   dragger!MouseUp}"
  l:onmouseover="{
   dragger!MouseOver(event.target))}">
 { for (var item <- items) <li>{ item }</li> }
 </ul>
}

<html>
 <body>
  <h1>Draggable lists</h1>
  <h2>Great Bears</h2>
  {
   draggableList("bears",
    ["Pooh", "Paddington", "Rupert", "Edward"]
   )
  }
  <h2>Great Beers</h2>
  {
   draggableList("beers",
    ["Guinness", "Stella Artois", "McEwan"]
   )
  }
 </body>
</html>
```

Figure 11. Draggable lists in Links

the client while continuing to run on the server, terminating the computation if the client does not respond to progress updates after a reasonable interval. This is not possible currently, because the design of Links deliberately limits the ways in which the server can communicate with the client, in order to guarantee that long-term session state is maintained on the client rather than the server. This is not appropriate in all circumstances, and future work on Links will need to consider a more general design.

3. Client-Server Computing

A Links program can be seen as a distributed program that executes across two locations: a client (browser) and a server. The programmer can optionally annotate a function definition to indicate where it should run. Functions on the client can invoke functions on the server, and vice-versa.

This symmetry is implemented on top of the asymmetric mechanisms made available by web browsers and servers. Current standards only permit the browser to invoke a function on the server. The server can return a value to the browser when done, but there is no provision for the server to invoke a function on the browser directly.

Our implementation is also *scalable*, in the sense that session state is preserved in the client rather than the server. Since server resources are at a premium in the web environment, our implemen-

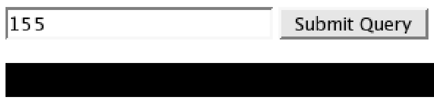
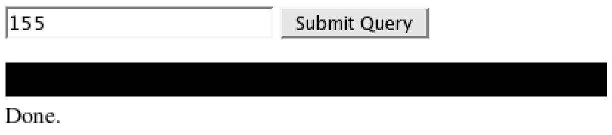


Figure 12. Progress bar



Done.

Figure 13. Progress bar displaying the final answer

```
fun compute(count, total) server {
 if (count < total) {
  showProgress(count, total);
  compute(count+1, total)
 } else "done"
}

fun showProgress(count, total) client {
 var percent = 100.0 *. count /. total;
 domReplaceElement(
  <div id="bar"
       style="width:floatToString(percent)%;
              background-color: black"/>,
  domGetRefById("bar")
 );
}

fun showAnswer(answer) client {
 domReplaceNode(
  <div id="bar">{stringToXml(answer)}</div>
  domGetRefById("bar")
 );
}

<html>
 <body id="body">
  <form l:onsubmit=
    "{showAnswer(compute(0,stringToInt(n)))}">
   <input type="text" l:name="{n}"/>
   <input type="submit"/>
  </form>
  <div id="bar"/>
 </body>
</html>
```

Figure 14. Progress bar in Links

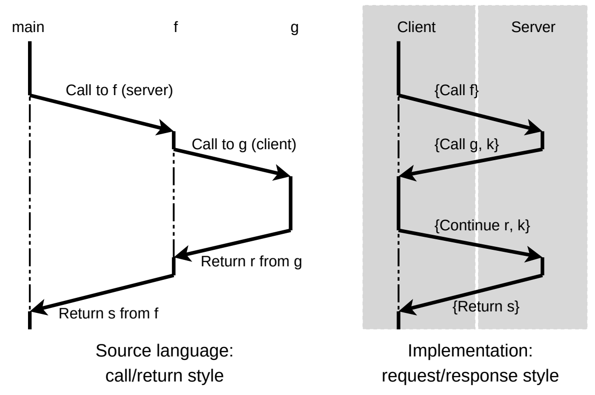


Figure 15. Semantic behavior of client/server annotations

tation requires no server resources unless the server is actively doing work.

We achieve this by using a generalization of continuation-passing style, which we call *resumption-passing style*. It is now common on the web to use continuations to "invert back the inversion of control" [48], permitting a process server to retain control after sending a form to the client. Here, we permit a process on the server to retain control after invoking an arbitrary function on the client.

Figure 15 shows how the call/return style of programming offered by Links differs from the standard request/response style, and how request/response style can emulate the call/return style. The left-hand diagram shows a sequence of calls between functions annotated with "client" and "server." The solid line indicates the active thread of control as it descends into these calls, while the dashed line indicates a stack frame which is waiting for a function to return. In this example, `main` is a client function which calls a server function `f` which in turn calls a client function `g`. The semantics of this call and return pattern are familiar to every programmer.

The right-hand diagram shows the same series of calls as they are implemented. The dashed line here indicates that some local storage is being used as part of this computation. During the time when `g` has been invoked but has not yet returned a value, the server stores nothing locally, even though the language provides an illusion that `f` is "still waiting" on the server. All of the server's state is encapsulated in the value `k`, which it passed to the client with its call.

To accomplish this, the program is compiled to two targets, one for each location, and begins running at the client. In this compilation step, server-annotated code is replaced on the client by a remote procedure call to the server. This RPC call makes an HTTP request indicating the server function and its arguments. The client-side thread is effectively suspended while the server function executes.

Likewise, on the server side, a client function is replaced by a stub. This time, however, the stub will not make a direct call to the client, since in general web browsers are not addressable by outside agents. However, since the server is always working on behalf of a client request, we have a channel on which to communicate. So the stub simply gives an HTTP response indicating the server-to-client call, along with a representation of the server's continuation, to be resumed when the server-to-client call is complete. Upon returning in this way, all of the state of the Links program is present at the client, so the server need not store anything more.

When the client has completed the server-to-client call, it initiates a new request to the server, passing the result and the server continuation. The server resumes its computation by applying the continuation.

3.1 Client-side Concurrency

A Links program is also concurrent. A program executing on the client can spawn new processes, which can communicate by passing messages. If any process makes a call to the server, it is seen as moving its place of execution to the server; it can move again if this call returns, or upon a server-to-client call. Client processes will keep executing on the client while one of them calls the server. Also, multiple client processes can make simultaneous remote calls.

Our client-side compilation target (JavaScript) is not inherently concurrent. To implement concurrency, we apply the following techniques:

- Compiling to continuation-passing style,
- Inserting explicit context-switch instructions,
- Eliminate the stack between context switches with `setTimeout`,
- Asynchronous remote calls using `XMLHttpRequest`.

Producing JavaScript code in CPS allows us to reify each thread's control state. To switch threads, we can insert "yield" instructions in the compiled code; "yield" adds the current continuation to a pool, chooses a new one from the pool, and runs it. However, typical JavaScript implementations do not use tail-call elimination, and have a severely limited call stack. To get around this, we use the `setTimeout` feature of the JavaScript VM. This function adds a closure to the VM's set of timeout callbacks; such a callback will be invoked when its timeout period (in milliseconds) has expired, and no other JavaScript is running, i.e., whenever some code has returned to the top level. Since our code is in CPS, we can return to the top level (after registering the current continuation as a callback) without losing control state.

Returning to the top level also allows other code registered with the VM to run: for example, event handlers given in attributes like `onclick` and `onmouseout`, as well as the callbacks that are associated with RPC requests.

In concurrent Links, client-to-server calls are implemented using the asynchronous form of `XMLHttpRequest`, an API for making HTTP requests, which all popular browsers support. To use the asynchronous form of `XMLHttpRequest`, we provide a callback function which invokes the continuation of the calling thread.

3.2 Related work: continuations for web programming

The idea of using continuations as a language-level technique to structure web programs has been discussed in the literature [47, 48] and used in several web frameworks (such as Seaside, Borges, PLT Scheme, RIFE and Jetty) and applications, such as ViaWeb's e-commerce application [22] which became Yahoo! Stores. Most these take advantage of a call/cc primitive in the source language, although some implement a continuation-like object in other ways. Each of these systems creates URLs which correspond to continuation objects. The most common technique for establishing this correspondence is to store the continuation in memory, associated with a unique identifier which is included as part of the URL.

Relying on in-memory tables makes such a system vulnerable to system failures and difficult to distribute across multiple machines. Whether stored in memory or in a database, the continuations can require a lot of storage. Since URLs can live long in bookmarks, emails, and other media, it is impossible to predict for how long a continuation should be retained. Most of the above frameworks destroy a continuation after a given period of time. Even with a modest lifetime, server storage needs can be quite high, as each request may generate many continuations and there may be many simultaneous users.

Our approach differs from these implementations by following the approach described recently by the PLT Scheme team [24]. Rather than storing a continuation object at the server and referring to it by ID, we serialize continuations and embed them completely in URLs and hidden form variables. To make this efficient, we assume that the program itself is fixed over time. Then we can defunctionalize the continuation object ([49]), replacing functions with unique identifiers. The resulting representation only identifies the closures which need to execute in the future of the computation, and the dynamic data needed by those closures.

4. Database

Links provides language-level support for accessing databases. Database tables are presented to the Links programmer as collections of records. The implementation compiles code that processes data stored in tables into the appropriate query language. (The current version of Links supports SQL as a target language; future versions will also support XQuery.)

Basic data sources may be referred to by the name of the corresponding SQL table. The declaration of the data source in the program must give the types of each field accessed, in order to support compile-time checking. (In a future version, the compiler will connect with the database to obtain this data from its schema instead.)

For example, consider the following nested comprehension:

```
for (var b <- books)
for (var a <- authors)
where (b.isbn == a.isbn && b.year < 2000)
 [(title=b.title, author=a.author)]
```

Everything in this code is expressible in SQL, so Links translates the entire fragment to an SQL query:

```
select b.title, a.author
from books as b, authors as a
where b.isbn = a.isbn AND b.year < 2000
```

In contrast, if we introduce a call to a function whose definition is not visible to the compiler, Links can only translate part of the comprehension to SQL; the other part is executed in memory:

```
for (var b <- books)
for (var a <- authors)
where (b.isbn == a.isbn && odd(b.year))
 [(title=b.title, author=a.author)]
```

Links translates this second example to the following combination of SQL and Links code:

```
for (var (title, year, author)
             <- select b.title, b.year, a.author,
                from books as b, authors as a
                where b.isbn = a.isbn)
 where (odd(year))
   [(title=title, author=author)]
```

4.1 Optimization

The Links optimizer is structured as of a set of rewrite rules over the syntax tree. A subset of the rules that transform Links code into database queries is shown in Figure 16. A query, for the purpose of these rules, is written in the form

```
select COLS
from TABS
where CONDS
```

representing a set of rows to be retrieved from some joined tables *TABS*, where only rows satisfying the conditions *CONDS* and only columns listed in *COLS* are returned. We write ($\bar{f} = \bar{x}$) to abbreviate ($f_1 = x_1 \ldots, f_n = x_n$), and similarly for other uses of overbar. If $\bar{f}$ is the sequence of fields $f_1, \ldots, f_n$ and π is a selection of indices $i_1, \ldots, i_k$ then $\bar{f}_\pi$ is the subsequence $f_{i_1}, \ldots, f_{i_k}$.

The first rule, *join*, transforms a nested comprehension into a single loop whose query is formed by cross-joining the queries of the inner and outer loops. The *select* rule lifts *where*-conditions of a comprehension into a query, conjoining them with the conditions of the query. Any free variables in the lifted condition will receive values from the Links environment at the time the query is evaluated. The third rule, *project*, limits the columns of the query to those bound fields that are actually mentioned in the body of the comprehension. The two auxiliary rules, *normalize* and *rename*, ensure that row destructuring is performed in the binding section of the comprehension (leaving no projections in the body) and that there are no clashes between the names of columns, respectively. Kleisli uses fairly complex rules for translation into SQL; the rules given here are significantly simpler, and we believe yield as good a translation.

Many standard operations on sequences have counterparts in SQL. The functions `take` and `drop` are of particular interest. The use of these functions may reveal that the program only makes use of some subset of the results of a query, in which case it is only necessary to retrieve that subset from the database, greatly reducing the computational load. For example, in the dictionary suggest example of Section 2.2, only the first ten results of each query are used by the program. In a similar way a combination of `take` and `drop` can be used for paginating a large result set.

Our optimization rules are interdependent, and the order in which they are applied cannot be arbitrarily varied without affecting correctness. For instance, the rules in Figure 16 must not be applied after the rules in Figure 17; a query which has a limit clause cannot, in general, be safely extended with additional tables and conditions.

Optimization rules for moving `take` and `drop` operations into queries are shown in Figure 17. For the purpose of these rules we extend the query definition with two additional elements: `limit`, which ranges over $\mathbb{N}_\infty$ and has a default value of ∞, and `offset`, which ranges over $\mathbb{N}$ and has a default value of 0. These correspond to the (non-standard, but widely available) SQL qualifiers of the same names. The *take* rule restricts the number of rows retrieved to at most `i`; the *drop* rule modifies the query to discard the first `i` rows.

The same technique is used to translate sort operations on collections of rows into the `ORDER BY` clause of SQL queries. There are several similar optimization opportunities that Links does not exploit, such as translating so-called aggregate functions. Furthermore, query optimization is currently hindered by the fact that the Links compiler is missing a number of standard optimizations, such as inlining of function calls. We expect to address these defects in a future version of the optimizer.

4.2 Implementation

Following Kleisli, the Links optimizer is implemented using rules and strategies. Each rule transforms syntax tree nodes that match a particular pattern; at each node a rule application either succeeds, yielding a new subtree to replace the node, or fails, resulting in no change to the tree. Strategies are combinators for rules. We use strategies to control the order of rule applications; some *join*

```
    for (var (f̄=x̄) <- select COLS
                       from TABS
                       where CONDS)
    for (var (ḡ=ȳ) <- select COLS′
                       from TABS′
                       where CONDS′)
      e
  ⟹
    for (var (f̄,ḡ=x̄,ȳ) <- select COLS,COLS′
                           from TABS, TABS′
                           where CONDS and CONDS′)
      e
```

select

```
    for (var (f̄=x̄) <- select r̄.ḡ as f̄
                       from TABS
                       where CONDS)
    where (c)
      e
  ⟹
    for (var (f̄=x̄) <- select r̄.ḡ as f̄
                       from TABS
                       where CONDS and c[r̄.ḡ/x̄])
      e
```

project

```
    for (var (f̄=x̄) <- select r̄.ḡ as f̄
                       from TABS
                       where CONDS)
      e
  ⟹
    for (var (f̄π=x̄π) <- select r̄π.ḡπ as f̄π
                         from TABS
                         where CONDS)
      e
```

where π is such that $\bar{x}_\pi = fv(e) \cap \{\bar{x}\}$

normalize

```
    for (var z <- select r̄.ḡ as f̄
                   from TABS
                   where CONDS)
      e
  ⟹
    for (var (f̄=x̄) <- select r̄.ḡ as f̄
                       from TABS
                       where CONDS)
      e[(f̄=x̄)/z]
```

rename

```
    for (var (f̄=x̄) <- select r̄.ḡ as f̄
                       from TABS
                       where CONDS)
      e
  ⟹
    for (var (f̄′=x̄) <- select r̄.ḡ as f̄′
                        from TABS
                        where CONDS)
      e
```

Figure 16. Optimization rewrite rules

take

```
    take(i, select Q limit m offset n)
  ⟹ select Q limit min(m,i) offset n
```

drop

```
    drop (i, select Q limit ∞ offset n)
  ⟹ select Q limit ∞ offset n+i
    drop(i, select Q limit m offset n)
  ⟹ select Q limit m−i offset n+i
```

Figure 17. Optimization rules for *take* and *drop*

`never`	a rule that always fails
`both a b`	a rule that applies a and then b
`either a b`	a rule that applies b iff a fails
`after a b`	a rule that applies b iff a succeeds
`several`	n-ary version of both
`any`	n-ary version of either
`sequence`	n-ary version of after
`topdown r`	apply r to a node, then 'topdown r' to its children
`bottomup r`	apply 'bottomup r' to a node's children, then r to the node
`topdown1 r`	as topdown, but with at most one node rewrite in each subtree
`bottomup1 r`	as bottomup, but with at most one node rewrite in each subtree
`iterate`	repeatedly apply a rule until it fails

Figure 18. Rewriting strategy combinators

such mechanism is necessary because our rules are not confluent. Our strategies, listed in Figure 18, apply rules to entire trees, or provide ways of combining and sequencing rules. For example, `topdown (both a b)` is a rewrite rule that applies the rules `a` and `b` in sequence to all the nodes of a tree in a top-down fashion.

In contrast to the implementation of Kleisli, applying a strategy to a rule does not change its type, resulting in enhanced compositionality. In Kleisli, the top-down strategy has the type `RULEDB -> SYN -> SYN`: it accepts a set of rules and configuration variables and returns a rewriter for syntax trees. Our `topdown` has type `Rewriter -> Rewriter`; applying `topdown` to a rewriter (e.g. a rule) yields a new rewriter that can be further composed and transformed using the other combinators. The use of strategies (such as `either` and `after`) rather than mutable state to conditionally sequence rules is an additional improvement in our implementation.

5. Conclusions and future work

We have demonstrated that it is possible to design a single language in which one can program all three tiers of a web application. This eliminates the usual impedance mismatch and other problems connected with writing an application distributed across multiple languages and computers. We have shown that we can support Ajax-style interaction in the client, programming applications such as dictionary suggest, draggable lists, and a progress bar. Our programming language compiles to JavaScript to run in the client, and SQL to run on the database.

Our results are preliminary, and much remains to be done.

The interface for passing values from HTML forms to the application logic (using `l:name`) is rather low-level. For instance, one would like to have a single abstraction that represents a form for entering a date, where the abstraction would present the same interface regardless of whether the date is entered as a string in a form field; by selecting values from three drop-down menus (for date, month, and year); or by clicking on an image of a calendar. This does not appear easy to do with the current system. For the future, we would like to develop a library that offers a more abstract approach to forms, perhaps along the lines used in WASH [54] or iData [45].

Database access is crucial. We need to measure the performance of our compiled SQL, and ensure that reasonable SQL is generated across a wide range of uses. Tuning of SQL is a widespread phenomenon, and it is not yet clear to what extent we need to support tuning, and how tuning can be supported in a way compatible with the Links approach. One of the advantages of Kleisli is that it can support multiple data sources, such as ASN.1 and other semi-structured data sources in addition to SQL data, and it provides good support for data integration. We intend to examine supporting other data sources with Links, such as XQuery for XML data. We also need to look into support for transactions.

Performance is also crucial, as successful web applications may run on multiple servers and support thousands of users. We currently have performed only small scale experiments, and the current interpreter is not particularly fast. We need to engineer better performance and devise experiments to test how the system scales.

Our current model of concurrency is limited because we insist on keeping all session state in the client. We need to explore other system models, including how to support concurrency on the server and how to support systems with multiple clients and servers.

Security is vital. Our current implementation completely exposes the state of the application to the client. A better system needs to cryptographically encode any sensitive data when it is present on the client. Web applications may wish to choose among a number of different means of storing data, each with different scopes and different security properties (e.g., in the session state on the client, in a cookie on the client that persists across sessions, or in the database; and using various encryption techniques). We need to provide some uniform framework into which we can slot different mechanisms for storing data and providing security.

Ultimately, we would like to grow a user community for Links. This depends on a number of factors, many of which are not well understood. One important factor seems to be the availability of good libraries. Good interfaces to other systems may help to rapidly develop useful functionality. A vexing question here is how to access facilities of languages that take a mostly imperative view of the world without vitiating the mostly functional approach taken by Links.

To make Links truly successful will require bringing together researchers and developers from many localities, many perspectives, and many communities. We would like to join our efforts with those of others — let us know if you are interested!

References

[1] J. Armstrong, M. Williams, R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[2] D. L. Atkins, T. Ball, G. Bruns and K. C. Cox. Mawl: A domain-specific language for form-based services. Software Engineering, 25(3):334 346, 1999.

[3] J. Armstrong. Concurrency oriented programming in Erlang. Invited talk, FFG 2003.

[4] K. Beck. *Extreme Programming Explained*. Addison Wesley, 2000.

[5] C. Brabrand, A. Møller, M. Ricky, M. Schwartzbach. PowerForms: Declarative client-side form field validation. *World Wide Web Journal* 3(4), 2000.

[6] C. Brabrand, A. Møller, M. Schwartzbach. The Bigwig project. *TOIT*, 2(2), 2002.

[7] N. Benton, L. Cardelli, C. Fournet. Modern concurrency abstractions for C♯. *TOPLAS*, 26(5), 2004.

[8] N. Benton, A. Kennedy, and C. Russo. Adventures in interoperability: the SML.NET experience. *PPDP*, 2004.

[9] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. *ICFP*, 2003.

[10] G. Bierman, E. Meijer, W. Schulte. Programming with rectangles, triangles, and circles. *XML Conference*, 2003.

[11] H. Böttger, A. Møller, M. Schwartzbach. Contracts for cooperation between web service programmers and HTML designers. Tech report, BRICS, 2003.

[12] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *TCS*, 149(1), 1995.

[13] R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. *Lisp Conference*, 1980.

[14] A. Christensen, A. Møller, M. Schwartzbach. Extending Java for high-level web service construction. *TOPLAS*, 25(6), 2003.

[15] A. Christensen, A. Møller, M. Schwartzbach. Precise analysis of string expressions. *SAS*, 2003.

[16] L. Damas and R. Milner. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*. 1982.

[17] S. El-Ansary, D. Grolaux, P. Van Roy, M. Rafea. Overcoming the multiplicity of languages and technologies for web-based development. *Mozart/Oz Conference*, LNCS 3389, 2005.

[18] M. Fernández, J. Siméon, P. Wadler. Introduction to the XQuery Formal Semantics. Katz, ed, *XQuery for Experts*, Addison-Wesley, 2004.

[19] C. Fournet, G. Gonthier. The Join Calculus: a language for distributed mobile programming. *Applied Semantics*, LNCS 2395, 2002.

[20] C. Fournet, F. Le Fessant, L. Maranget, A. Schmitt. JoCaml: a language for concurrent distributed and mobile programming. *Advanced Functional Programming*, LNCS 2638, 2003.

[21] J. Garret. Ajax: a new approach to web applications. 2005.

[22] P. Graham. "Method for client-server communications through a minimal interface." United States Patent no. 6,205,469. March 20, 2001.

[23] P. Graham. Beating the averages. 2001.

[24] P. Graunke, R. B. Findler, S. Krishnamurthi, M. Felleisen. Automatically restructuring programs for the web. *ASE*, 2001.

[25] P. Graunke, R. B. Findler, S. Krishnamurthi, M. Felleisen. Modeling web interactions. *ESOP*, 2003.

[26] P. Graunke, S. Krishnamurthi, S. van der Hoeven, M. Felleisen. Programming the web with high-level programming languages. *ESOP*, 2001.

[27] V. Gapeyev, M. Levin, B. Pierce, A. Schmitt. The Xtatic experience. *PLAN-X*, 2005.

[28] T. Grust. Monad comprehensions, a versatile representation for queries. In P. Gray, *et al* (editors), *The Functional Approach to Data Management*, Springer Verlag, 2003.

[29] J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[30] H. Hosoya, B. Pierce. XDuce: A typed XML processing language. *WebDB*, LNCS 1997, 2000.

[31] Haruo Hosoya, Benjamin C. Pierce. XDuce: A typed XML processing language. *TOIT*, 3(2), 2003.

[32] IMRG. Retail's New Order. Press release, 2003.

[33] C. Kirkegaard and A. Møller. *Type Checking with XML Schema in Xact*. Presented at Programming Language Technologies for XML, PLAN-X '06. September, 2005

[34] L. Libkin, L. Wong. Query languages for bags and aggregate functions. *JCSS*, 55(2):241–272, 1997.

[35] Microsoft Corporation. *DLinq: .NET Language Integrated Query for Relational Data* September 2005

[36] R. Milner. A Theory of Type Polymorphism in Programming. J.

Comput. Syst. Sci. 17(3): 348-375 (1978)

[37] M. MacHenry, J. Matthews. Topsl: a Domain-Specific Language for On-Line Surveys. *Scheme Workshop*, 2004.

[38] S. Marlow, P. Wadler. A practical subtyping system for Erlang. *ICFP*, 1997.

[39] A. Møller, M. Schwartzbach. The design space of type checkers for XML transformation languages. *ICDT*, LNCS 3363, 2005.

[40] G. Narra. ObjectGraph Dictionary. `http://www.objectgraph.com/dictionary/how.html`

[41] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. *POPL*, 2005.

[42] M. Carlsson, J. Nordlander, D. Kieburtz. The semantic layers of Timber. *ASPLAS*, 2003.

[43] M. Odersky, V. Cremet, C. Rockl, M. Zenger. A nominal theory of objects with dependent types. *ECOOP*, 2003.

[44] M. Odersky *et al*. An overview of the Scala programming language. Technical report, EPFL Lausanne, 2004.

[45] R. Plasmeijer and P. Achten. iData For The World Wide Web: Programming Interconnected Web Forms. *FLOPS*, 2006.

[46] F. Pottier and D. Rémy. The essence of ML type inference. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[47] C. Queinnec. Continuations to program web servers. *ICFP*, 2000.

[48] Christian Queinnec *Inverting back the inversion of control or, continuations versus page-centric programming*, SIGPLAN Not., 2003

[49] J. Reynolds. *Definitional interpreters for higher-order programming languages*. ACM '72: Proceedings of the ACM annual conference, 1972.

[50] A. Sandholm, M. I. Schwartzbach. A type system for dynamic web documents. *POPL*, 2000.

[51] Science for global ubiquitous computing. A fifteen-year grand challenge for computing research. UKCRC, 2003.

[52] J. Simeon, P. Wadler. The essence of XML. *POPL*, 2003.

[53] D. Syme. F♯ web page. `research.microsoft.com/projects/ilx/fsharp.aspx`

[54] P. Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. *PADL*, 2002.

[55] P. Van Roy, *et al*. Logic programming in the context of multiparadigm programming: the Oz experience. *TLP* 3(6):717–763, November 2003.

[56] P. Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science, Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 2:461-493, 1992.

[57] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. *Workshop on Standard ML*, 1998.

[58] P. Wadler and P. Thiemann. The marriage of effects and monads. *TOCL*, 4(1), 2003.

[59] C. Wikström and H. Nilsson. Mnesia – an industrial DBMS with transactions, distribution, and a logical query language. *CODAS*, 1996.

[60] L. Wong. Kleisli, a functional query system. *JFP*, 10(1), 2000.

[61] XML Query and XSL Working Groups. XQuery 1.0: An XML Query Language, W3C Working Draft, 2005.