Links: Web Programming Without Tiers*

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop

University of Edinburgh

Abstract. Links is a programming language for web applications that 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 supports rich clients running in what has been dubbed *'Ajax'* style, and supports concurrent processes with statically-typed message passing. Links is *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. Client-side concurrency in JavaScript and transfer of computation between client and server are both supported by translation into continuation-passing style.

1 Introduction

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 pages 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 pages described 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.

Increasingly, web applications designers are migrating work into the browser. "Rich client" systems, such as Google Mail and Google Maps, use a new style of interaction dubbed "*Ajax*" [11]. Client-side Links 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 standard and reliable versions of JavaScript will become available over the next few years.

Links is a strict, typed, functional language. It incorporates ideas proven in other functional languages, including:

– database query optimization, as found in Kleisli and elsewhere,

* Submitted to FMCO '06

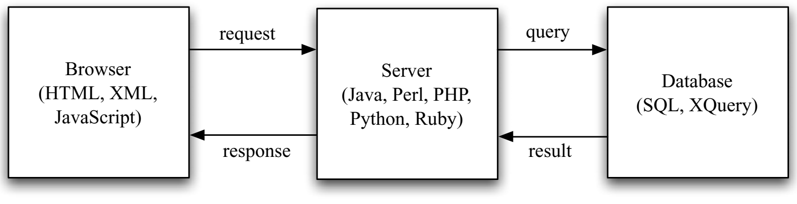


Fig. 1. Three-tier model

– continuations for web interaction, as found in PLT Scheme and elsewhere, and
– concurrency with message passing, as found in Erlang and elsewhere,

All three of these features work better with immutable values rather than mutable objects. In Links, side effects play a limited (though important) role, being used for updates to the database and display, and communication between concurrent processes. 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.

Links programs are *scalable* in the sense that session state is preserved in the client rather than the server. Many commercial web tools (like J2EE) and most research web tools (including current releases of PLT Scheme [15] and Mozart QHTML [9]) are not scalable in this sense.

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.

Database programming. Queries are written in the Links notation and compiled into SQL, a technique pioneered by Kleisli [7, 35] and now used in LINQ [18].

Web interaction. The notion that a programming language could provide support for web interaction first appears in the programming language MAWL [1]. 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 [25], Graham [13] (in a commercial system sold to Yahoo and widely used for building web stores), Felleisen and others [14, 15], and Thiemann [31].

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 [2] and Mozart [32].

XML programming. Links provides convenient syntax for constructing XML data, similar to that provided by XQuery [36]. The current version does not support regular expression types, as found in XDuce and other languages, but we may add them in a future version. Regular expression types were given a low priority, because they are already well understood from previous research [19].

Other languages. Other languages for web programming include Xtatic [16], Scala [22], Mozart [9], SML.NET [5], F♯ [30], Cω (based on Polyphonic C♯ [4] and Xen [6]), HOP [29], Ocsigen [3]. 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 as an alternative to the current multiplicity of languages will be a principal attraction of Links.

This paper. We introduce Links by describing three examples in Section 2. Section 3 sketches our implementation of concurrency and client-server interaction. Section 4 gives an SQL-compilable subset of Links and details how it is compiled into SQL, while Section 5 describes typing for processes. Section 6 discusses some shortcomings of the current implementation and Section 7 concludes.

2 Links by example

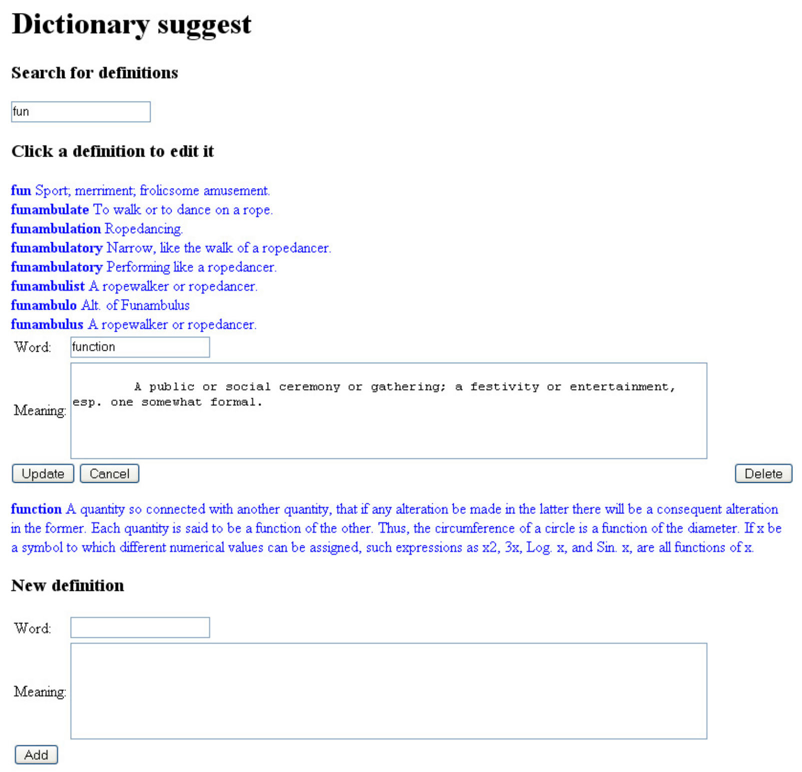
This section introduces Links by a series of examples. The reader is encouraged to try these examples online at

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

We begin with a dictionary-suggest example to introduce the basic functionality of Links, and then present two further examples that demonstrate additional capabilities: a draggable list, and a progress bar.

2.1 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 2). In addition, the user can add, update and delete definitions. To add a new definition the user fills in and submits the form at the bottom of the page by clicking 'Add'. To update an existing definition the user clicks on one of the suggestions, which replaces the suggestion with a form containing the existing definition, and then edits and submits the form by clicking 'Update'. This form also includes buttons for cancelling the update (which restores the original suggestion) and deleting the definition.



Dictionary suggest

Search for definitions

fun

Click a definition to edit it

fun Sport; merriment; frolicsome amusement.
funambulate To walk or to dance on a rope.
funambulation Ropedancing.
funambulatory Narrow, like the walk of a ropedancer.
funambulatory Performing like a ropedancer.
funambulist A ropewalker or ropedancer.
funambulo Alt. of Funambulus
funambulus A ropewalker or ropedancer.

Word:	function
Meaning:	A public or social ceremony or gathering; a festivity or entertainment, esp. one somewhat formal.

function A quantity so connected with another quantity, that if any alteration be made in the latter there will be a consequent alteration in the former. Each quantity is said to be a function of the other. Thus, the circumference of a circle is a function of the diameter. If x be a symbol to which different numerical values can be assigned, such expressions as x2, 3x, Log. x, and Sin. x, are all functions of x.

New definition

Word:	
Meaning:	

Fig. 2. Dictionary suggest

This application is of interest because it must perform a database lookup and update the display at every keystroke. Applications such as Google Suggest [17] have a similar structure.

The Links version is based on an ASP.NET version of the same application, available online [21], using the same dictionary and formatting. It extends the ASP.NET version by allowing the definitions to be added, updated and deleted. 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 Figures 3–5; following is a short walk-through of the code. On each keystroke, a `Suggest` message containing the current contents of the text field is sent to the `handler` process. The `handler` process passes the text content to the function `suggest`. 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. Doing the server interaction in a separate `handler` process allows the user interaction to remain responsive, even while looking up suggestions.

```
var defsTable =
 table "definitions" with
 (id:String, word:String, meaning:String)
 where id readonly from database "dictionary";

fun newDef(def) server { insert defsTable values [def] }
fun updateDef(def) server {
 update (var d <-- defsTable) where (d.id == def.id)
 set (word=def.word, meaning=def.meaning)
}
fun deleteDef(id) server {
 delete (var def <-- defsTable) where (def.id == id)
}

fun completions(s) server {
 if (s == "") [] else {
  take(10, for (var def <-- defsTable)
           where (def.word ~ /s.*/) orderby (def.word)
            [def])
 }
}

fun suggest(s) client {
 replaceChildren(format(completions(s)),
                 getNodeById("suggestions"))
}

fun editDef(def) client {
 redraw(
  <form l:onsubmit="{
   var def = (id=def.id, word=w, meaning=m); updateDef(def);
   redraw(formatDef(def), def.id)}" method="POST">
   <table>
    <tr><td>Word:</td><td>
     <input l:name="w" value="{def.word}"/></td></tr>
    <tr><td>Meaning:</td><td>
     <textarea l:name="m" rows="5" cols="80">
      {stringToXml(def.meaning)}</textarea></td></tr>
   </table>
   <button type="submit">Update</button>
   <button l:onclick="{redraw(formatDef(def), def.id)}">
    Cancel</button>
   <button l:onclick="{deleteDef(def.id); redraw([],def.id)}"
    style="position:absolute; right:0px">Delete</button>
  </form>,
  def.id)
}
```

Fig. 3. Dictionary suggest in Links (1)

```
fun redraw(xml, defId) client {
 replaceChildren(xml, getNodeById("def:"++defId))
}

fun formatDef(def) client {
 <span l:onclick="{editDef(def)}">
  <b>{stringToXml(def.word)}</b>
  {stringToXml(def.meaning)}<br/>
 </span>
}

fun format(defs) client {
 <#>
  <h3>Click a definition to edit it</h3>
  for (var def <- defs)
   <span class="def" id="def:def.id">{formatDef(def)}</span>
 </#>
}

fun addForm(handler) client {
 <form l:onsubmit="{handler!NewDef((word=w, meaning=m))}">
  <table>
  <tr><td>Word:</td><td>
   <input type="text" l:name="w"/></td></tr>
  <tr><td>Meaning:</td><td>
   <textarea l:name="m" rows="5" cols="80"/></td></tr>
  <tr><td><button type="submit">Add</button></td></tr>
  </table>
 </form>
}

var handler = spawn {
 fun receiver(s) {
  receive {
   case Suggest(s) -> suggest(s); receiver(s)
   case NewDef(def) ->
    newDef(def);
    replaceChildren(addForm(self()), getNodeById("add"));
    suggest(s); receiver(s)
  }
 }
 receiver("")
};
```

Fig. 4. Dictionary suggest in Links (2)

```
<html>
 <head>
  <style>.def {{ color:blue }}</style>
  <title>Dictionary suggest</title>
 </head>
 <body>
  <h1>Dictionary suggest</h1>
  <h3>Search for definitions</h3>
  <form l:onkeyup="{handler!Suggest(s)}">
   <input type="text" l:name="s" autocomplete="off"/>
  </form>
  <div id="suggestions"/>
  <h3>New definition</h3>
  <div id="add">{addForm(handler)}</div>
 </body>
</html>
```

Fig. 5. Dictionary suggest in Links (3)

The rest of the code is concerned with modifying the database. A form for adding definitions is created by the function `addForm`. Clicking 'Add' sends a `NewDef` message to the `handler` process containing a new definition. The `handler` process calls `newDef` to add the definition, resets the form and updates the list of suggestions (in case the new definition appears in the current list of suggestions).

Clicking on a definition invokes the function `editDef`, which calls the function `redraw` in order to replace the definition with a form for editing it. Clicking 'Cancel' reverses this operation. Clicking 'Update' or 'Delete' performs the corresponding modification to the definition by calling `updateDef` or `deleteDef` on the server, and then updates the list of suggestions by calling the function `redraw` on the client.

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. Low-level syntactic details become a matter of habit that can be hard to change: we conjecture that using the familiar `f(x,y)` in place of the unfamiliar `f x y` will significantly lower the barrier to entry of functional programming.

One difference from JavaScript syntax is that we do not use the keyword **`return`**, which is too heavy to support a functional style. Instead, we indicate return values subtly (perhaps too subtly), by omitting the trailing semicolon.

Types. Links uses Hindley-Milner type inference with row variables [24]. As basic types Links supports integers, floats, characters, booleans, lists, functions, records, and variants.

– A list is written `[`e_1`, ..., `e_k`]`, and a list type is written `[`A`]`.
– A lambda abstraction is written **`fun`** `(`x_1`, ..., `x_k`) {`e`}`, and a function type is written `(`A_1`, ..., `A_k`) ->` B.

- A record is written `(`f_1`=`e_1`, ..., `f_k`=`e_k`)`, and a record type is written `(`f_1`:`A_1`, ..., `f_k`:`A_k` | `r`)`, where r is an optional row variable. Field names for records begin with a lower-case letter.
- A variant is written F_i`(`e_i`)` and a variant type is written `[|`F_1`:`A_1`, ..., `F_k`:`A_k`|`r`|]`. Field names for variants begin with an upper-case letter.

Strings are lists of characters (as in Haskell), and tuples are records with natural number labels (as in SML). Apart from the table declaration in Figure 3, none of the examples in the paper explicitly mention types. This is partly because type inference renders type annotations unnecessary and partly to save space. Nevertheless, it should be stressed that all of the examples are type-checked statically, and static typing is an essential part of Links.

Links currently does not support any form of overloading; we expect to support overloading in future using a simplified form of type classes. As with regular expression types, 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 [31, 23].

XML. 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 supports syntactic sugar `<#> ... </#>` for specifying an XML forest literal as in the function *`format`* in Figure 4.

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. Links provides library functions to access and modify the DOM, based on the similar operations specified by the W3C DOM standard. Links supports two types for manipulating XML: `DomNode` is a mutable reference (similar to a *`ref`* type in ML), while `Xml` is an immutable list of trees. There is an operation that converts the former to the latter by making a deep copy of the tree rooted at the node, returning it in a singleton list. We expect eventually to support regular expression types for XML that refine each of these two types, and to support a notation like XPath for manipulating trees, but as these points are well understood (but a lot of work to implement) they are not a current priority.

Regular expressions. Matching a string against a regular expression is written `e ~ /r/` where *r* is a regular expression. Curly braces may be used to interpolate a string into a regular expression, so for example `/{s}.*/` matches any string that begins with the value bound to the variable `s`.

Interaction. The Links code specifies display of an XML document, the crucial part of which is the following:

```
<form l:onkeyup="{handler!Suggest(s)}">
 <input type="text" l:name="s" autocomplete="off"/>
</form>
<div id="suggestions"/>
```

The `l:name` attribute specifies that the string typed into the field should be bound to a string variable *s*.

The attributes `l:name` and `l:onkeyup` are special. The attribue `l:onkeyup` is followed by Links code in curly braces that is *not* immediately evaluated, but is evaluated whenever a key is released while typing in the form. (Normally, including curly braces in XML attributes, as elsewhere in XML, causes the Links code inside the braces to be evaluated and its value to be spliced into the XML.) The `l:name` attribute on an input field must contain a Links variable name, and that variable is bound to the contents of the input field.

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, `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. 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.

Our experience has shown that 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. We return to this point in the Section 6.

List comprehensions. The Links code calls the function *suggest* each time a key is released. This in turn calls *completions* to find the first ten completions of the prefix, and *format* to format the results as HTML for display. Both *completions* and *format* use **for** loops, in the former case also involving **where** and **orderby** clauses. These constructs correspond to what is written as a *list comprehension* in languages such as Haskell or Python. Each comprehension is equivalent to an ordinary expression using standard library functions, we give three examples, where the comprehension is on the left and its translation is on the right.

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

for (var x <- e1)                concatMap(
 where (e2)                       fun(x){if (e2) e3 else []},
  e3                              e1)

for (var x <- e1)                concatMap(
 orderby (e2)                     fun(x){e3},
  e3                              orderBy(fun(x){e2},e1))
```

Here `concatMap(f, xs)` applies function *f* to each element in list *xs* and concatenates the results, and `orderBy(f, xs)` sorts the list *xs* so that it is in ascending order after *f* is applied to each element. 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; indeed, currently **orderby** clauses only work reliably in code that compiles into SQL, because the absence of overloading means we have not yet implemented comparison on arbitrary types in the client or server).

Database. Links provides facilities to query and update SQL databases, where database tables are viewed as lists of records. We hope to provide similar facilities for XQuery databases in the future.

Links provides a **database** expression that denotes a connection to a database, and a **table** expression that denotes a table within a database. A database is specified by name (and optional configuration data, which is usually read from a configuration file), and a table is specified by the table name, the type signature of a row in the table, and the database containing the table.

The type of a table is distinct from the type of its list of records, since tables (unlike lists) may be updated. The coercion operation *asList* takes a table into the corresponding list, and `for (var x <-- e1) e2` (with a long arrow) is equivalent to `for (var x <- asList(e1)) e2` (with an ordinary arrow).

In the following example, there is a table of words, where each row contains three string fields, the word itself, its type (noun, verb, and so on), and its meaning (definition). Typically, one expresses queries using the Links constructs such as **for**, **where**, and **orderby**, and functions on lists such as *take* and *drop*. The Links compiler is designed to convert these into appropriate SQL queries over the relevant tables whenever possible. For example, the expression

```
take(10, for (var def <-- defsTable)
         where (def.word ~ /s.*/) orderby (def.word)
          [def])
```

compiles into the equivalent SQL statement

```
SELECT def.meaning AS meaning, def.word AS word
FROM definitions AS def
WHERE def.word LIKE '{s}%'
ORDER BY def.word ASC
LIMIT 10 OFFSET 0
```

This form of compilation was pioneered in systems such as Kleisli [7, 35], and is also central to Microsoft's new Language Integrated Query (LINQ) for .NET [18]. A related approach, based on abstract interpretation rather than program transformation, is described by Wiederman and Cook [34].

Note that the match operator ~ on regular expressions in Links is in this case compiled into the `LIKE` operator of SQL, and that the call to *take* in Links is compiled into `LIMIT` and `OFFSET` clauses in SQL. At run time, the phrase `{s}` in the SQL is replaced by the string contained in the variable *s*, including escaping of special characters where necessary. Links also contains statements to update, delete from, and insert into a database table, closely modeled on the same statements in SQL.

The `LIMIT` and `OFFSET` clauses are not part of the SQL standard, but are available in PostgreSQL, MySQL, and SQLite, the three targets currently supported by Links. For non-standard features such as this, Links can generate different code for different targets; for instance, it generates different variants of `INSERT` for PostgreSQL and MySQL.

Section 4 presents a subset of Links that is guaranteed to compile into SQL queries.

Concurrency. Links allows one to spawn new processes. The expression **`spawn`** `{e}` creates a new process and returns a fresh process identifier, the primitive `self()` returns the identifier of the current process, the command `e1 ! e2` sends message `e2` to the process identified by `e1`, and the expression

```
receive {
  case p1 -> e1
  ...
  case pn -> en
}
```

extracts the next message sent to the current process (or waits if there is no message), and executes the first case where the pattern matches the message. (Unlike Erlang, it is an error if no case matches, which fits better with our discipline of static typing.)

By convention, the messages sent to a process belong to a variant type. Event handlers are expected to execute quickly, so we follow a convention that each handler either sends a message or spawns a fresh process. For instance, in the dictionary suggest application, each event handler sends a message to a separate handler process that is responsible for finding and displaying the completions of the current prefix. This puts any delay resulting from the database lookup into a separate process, so any keystroke will be echoed immediately. Furthermore, the messages received by the handler process are processed sequentially, ensuring that the updates to the DOM happen in the correct order.

As well as any new processes spawned by the user, there is one distinguished process: the *main* process. The top-level program and all event handlers are run in the main process. Having all event handlers run in a single process allows us to guarantee that events are processed in the order in which they are received.

Client-server. The keyword **`server`** in the definition of `completions` causes invocations of that function to be evaluated on the server rather than the client, which is required because the database cannot be accessed directly from the client.

When the dictionary suggest application is invoked, the server does nothing except to transmit the Links code (compiled into JavaScript) to the client. If the user presses and releases a key then the `suggest` function will continue to run on the client as its definition is annotated with the keyword **`client`**. The client runs autonomously until `completions` is called, at which point the `XMLHttpRequest` primitive of JavaScript 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 `completions` 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.

Location annotations **`server`** and **`client`** are only allowed on top-level function definitions. If a top-level function definition does not have a location annotation then it

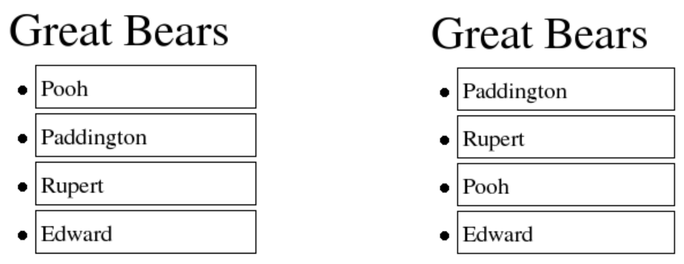


Fig. 6. Draggable list: before and after dragging

will be compiled for both the client and the server. A location annotation should be read as "this function *must* be run in the specified location". The implementation strategy for client-server communication is discussed further in Section 3.

2.2 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 Figure 6).

The code for draggable list is shown in Figure 7. 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 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.

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

```
fun waiting(id) {
 receive {
  case MouseDown(elem)     ->
   if (isElementNode(elem)
             && (parentNode(elem) == getNodeById(id)))
     dragging(id,elem)
   else waiting(id)
  case MouseUp             -> waiting(id)
  case MouseOver(newElem) -> waiting(id)
 }
}

fun dragging(id,elem) {
 receive {
  case MouseUp           -> waiting(id)
  case MouseDown(elem)   ->
   if (isElementNode(elem)
             && (parentNode(elem) == getNodeById(id)))
     dragging(id,elem)
  case MouseOut(toElem) ->
   if (isElementNode(toElem)
             && (parentNode(elem) == getNodeById(id)))
    {swapNodes(elem,toElem); dragging(id,elem)}
   else dragging(id,elem)
 }
}

fun draggableList (id,items) client {
 var dragger = spawn { waiting(id) };
 <ul id="{id}"
  l:onmouseup="{dragger!MouseUp}"
  l:onmouseuppage="{dragger!MouseUp}"
  l:onmousedown="{dragger!MouseDown(getTarget(event))}"
  l:onmouseout="{dragger!MouseOut(getToElement(event))}">
   { for (var item <- items) <li>{ item }</li> }
 </ul>
}

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

Fig. 7. Draggable lists in Links

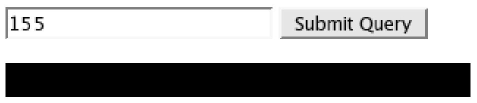
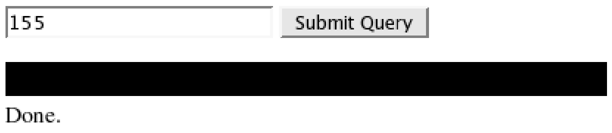


Fig. 8. Progress bar



Done.

Fig. 9. Progress bar displaying the final answer

2.3 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 8). When the computation is completed, the answer is displayed (see Figure 9).

The code for the progress bar application is shown in Figure 10. 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.

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 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

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

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

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

Fig. 10. Progress bar in Links

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.

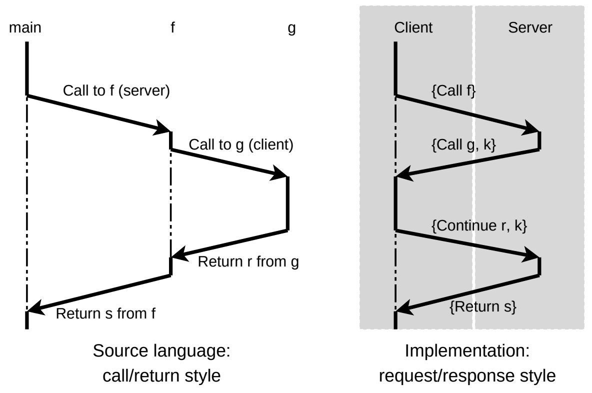


Fig. 11. Semantic behaviour of client/server annotations

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

We achieve this by using a variation 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" [26], 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 11 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 concurrent. We implement concurrency on the client, even though JavaScript does not contain primitives to support concurrency. Concurrency, client-to-server communication and server-to-client communication are implemented using the following techniques:

- compiling to continuation-passing style,
- inserting explicit context-switch instructions,
- registering asynchronous remote calls using `XMLHttpRequest`,
- eliminating the stack between context switches using either `setTimeout` or a trampoline.

Producing JavaScript code in CPS allows us to reify each process's control state. The compiler wraps every function application and every continuation application in special "yield" functions. After a certain number of calls to "yield" the current continuation is added to the JavaScript VM's collection of timeout callbacks allowing another continuation to run. This is implemented by calling the `setTimeout` primitive of JavaScript, which registers a callback to be invoked after a specified period of time.

The "yield" function for function application is shown in Figure 12.[1] The variable `_yieldGranularity` determines how often to yield to other processes. After every `_yieldGranularity` calls to `_yield` and `_yieldCont` control is relinquished using one of two methods.

- If execution is outside an event handler, then the built-in `setTimeout` function is invoked with a callback that sets the process id and then calls `f(a, k)`. Calling `setTimeout` allows other processes to run. The `_sched_pause` parameter specifies a lower bound on how long to wait before allowing this callback to run. For older browsers `_sched_pause` must be non-zero, otherwise

[1] The "yield" function for continuation application is the same except it takes as arguments continuation `k` and value `a`, and the function applications `f(a, k)` are replaced with continuation applications `k(a)`.

the callback may be run immediately and other callbacks may not get a chance to run. For newer browsers `_sched_pause` may be set to zero, as the callback is placed at the back of a queue of `setTimeout` callbacks. Typical JavaScript implementations do not use tail-call elimination, and have a severely limited call stack. Our use of `setTimeout` also sidesteps this limitation, since `setTimeout` discards the current stack. In our case the stack contains only tail calls that we need never return to, so this reclaims memory while retaining the desired behaviour. The timeout also provides an opportunity for the event handler or any callbacks from `XMLHttpRequest` to run.

– Inside an event handler execution is synchronous, allowing events to be handled atomically. In order to prevent event handlers from blocking concurrency, we recommend that they be short-lived (ideally just send a message or spawn a new process). Although event-handling is synchronous, it is still important to make sure that the browser does not run out of stack. Although running out of stack inside event handlers is unlikely to be a problem if programmers follow our guidelines, a *trampoline* is provided just in case. In order to yield, an exception is thrown containing the current continuation. Throwing the exception unwinds the stack. The trampoline catches the exception and then invokes the continuation. The trampoline is shown in Figure 13.

Unsurprisingly, the output of the CPS-translating compiler is slower than comparable direct style code. The performance penalty is significant but not disastrous: the total run-time of programs we measured was within an order of magnitude of equivalent hand-written JavaScript versions.

```
function _yield(f, a, k) {
  ++_yieldCount;
  if ((_yieldCount % _yieldGranularity) == 0) {
    if (!_handlingEvent) {
      var current_pid = _current_pid;
      setTimeout((function() {
                      _current_pid = current_pid;
                      f(a, k)}),
                 _sched_pause);
    }
    else {
      throw new _Continuation(function () { f(a,k) });
    }
  }
  else {
    return f(a,k);
  }
}
```

Fig. 12. The yield function

```
function _Continuation(v) { this.v = v }

function _wrapEventHandler(handler) {
  return function(event) {
    var active_pid = _current_pid;
    _current_pid = _mainPid;
    _handlingEvent = true;
    var cont = function () { handler(event) }
    for (;;) {
      try {
        cont();
        _handlingEvent = false;
        _current_pid = active_pid;
        return;
      }
      catch (e) {
        if (e instanceof _Continuation) {
           cont = e.v;
           continue;
        }
        else {
          _handlingEvent = false;
          _current_pid = active_pid;
          throw e;
        }
      }
    }
  }
}
```

Fig. 13. The event handler trampoline

Client-to-server calls are implemented using the asynchronous form of the function *XMLHttpRequest*, to which we pass a callback function that invokes the continuation of the calling process.

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 [25, 26] and used in several web frameworks (such as Seaside, Borges, PLT Scheme, RIFE and Jetty) and applications, such as ViaWeb's e-commerce application [12], 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 by the PLT Scheme team [14]. 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 ([27]), 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 Query compilation

A subset of Links expressions compiles precisely to a subset of SQL expressions. Figure 14 gives the grammar of the SQL-compilable subset of Links expressions and Figure 15 gives the grammar for the subset of SQL to which we compile.

All terms are identified up to α-conversion (that is up to renaming of bound variables, field names and table aliases). We allow free variables in queries, though they are not strictly allowed in SQL. Values for these variables will be supplied either at compile time, during query rewriting, or else at runtime. We use vector notation $\bar{v}$ to mean $\mathtt{v}_1, \ldots, \mathtt{v}_k$, where $1 \leq k$ and abuse vector notation in the obvious way in order to denote tuples, record patterns and lists of tables. For uniformity, we write an empty 'from' clause as **from** •. In standard SQL the clause would be omitted altogether. We assume a single database *db*, and write **table** *t* **with** $(f_1, \ldots, f_k)$ for *asList*(**table** *t* **where** $(f_1{:}A_1, \ldots, f_k{:}A_k)$ **from** *db*).

(expressions)	*e*	::=	*take*(*n*, *e*) | *drop*(*n*, *e*) | *s*
(simple expressions)	*s*	::=	**for** (*pat* <- *s*) *s*
		|	**let** *x* = *b* **in** *s*
		|	**where** (*b*) *s*
		|	**table** *t* **with** $\bar{f}$
		|	[($\bar{b}$)]
(basic expressions)	*b*	::=	b_1 *op* b_2
		|	**not** *b*
		|	*x*
		|	*lit*
		|	*z*.*f*
(patterns)	*pat*	::=	*z* | ($\bar{f}$=$\bar{x}$)
(operators)	*op*	::=	**like** | > | = | < | <> | **and** | **or**
(literal values)	*lit*	::=	**true** | **false** | *string-literal* | *n*
(finite integers)	*i*, *m*, *n*		
(field names)	*f*, *g*		
(variables)	*x*, *y*		
(record variables)	*z*		
(table names)	*t*		

Fig. 14. Grammar of an SQL-compilable subset of Links.

(queries)	*q*	::=	**select** *Q* **limit** *ninf* **offset** *n*
(query bodies)	*Q*	::=	*cols* **from** *tables* **where** *c*
(column lists)	*cols*	::=	$\bar{c}$
(table lists)	*tables*	::=	$\bar{t}$ **as** $\bar{a}$ | •
(SQL expressions)	*c*, *d*	::=	*c* op *d*
		|	**not** *c*
		|	*x*
		|	*lit*
		|	*a*.*f*
(integers)	*ninf*	::=	*n* | ∞
(table aliases)	*a*		

Fig. 15. The SQL grammar that results from query compilation.

TABLE

table t **with** $\bar{f}$
$\longrightarrow$ (a is fresh)
select $a.\bar{f}$ **from** t as a **where true limit** ∞ **offset** 0

LET

let x = b **in** s $\longrightarrow$ s[b/x]

TUPLE

[($\bar{b}$)] $\longrightarrow$ **select** $\bar{b}$ **from** • **where true limit** ∞ **offset** 0

JOIN

for (($\bar{f}$=$\bar{x}$) <-
  select $\bar{c}_1$ **from** $\bar{t}_1$ **as** $\bar{a}_1$ **where** d_1 **limit** ∞ **offset** 0)
    select $\bar{c}_2$ **from** $\bar{t}_2$ **as** $\bar{a}_2$ **where** d_2 **limit** ∞ **offset** 0
$\longrightarrow$ ($\bar{a}_1$ and $\bar{a}_2$ are disjoint)
select $\bar{c}_2[\bar{c}_1/\bar{x}]$ **from** $\bar{t}_1$ **as** $\bar{a}_1$, $\bar{t}_2$ **as** $\bar{a}_2$
  where d_1 **and** $d_2[\bar{c}_1/\bar{x}]$ **limit** ∞ **offset** 0

WHERE

where (b) (**select** $\bar{c}$ **from** $\bar{t}$ **as** $\bar{a}$ **where** d **limit** m **offset** n)
$\longrightarrow$
select $\bar{c}$ **from** $\bar{t}$ **as** $\bar{a}$ **where** d **and** b **limit** m **offset** n

RECORD

for (z <- **select** $\bar{c}$ **from** $\bar{t}$ **as** $\bar{a}$ **where** d **limit** m **offset** n) s
$\longrightarrow$ ($\bar{x}$ not free in s)
for (($\bar{f}$=$\bar{x}$) <- **select** $\bar{c}$ **from** $\bar{t}$ **as** $\bar{a}$
    where d **limit** m **offset** n) $s[\bar{x}/z.\bar{f}]$

TAKE

take(i, **select** Q **limit** m **offset** n)
$\longrightarrow$
select Q **limit** *min*(m, i) **offset** n

DROP

drop(i, **select** Q **limit** m **offset** n)
$\longrightarrow$
select Q **limit** $m-i$ **offset** $n+i$

DROP$_\infty$

drop (i, **select** Q **limit** ∞ **offset** n)
$\longrightarrow$
select Q **limit** ∞ **offset** $n+i$

Fig. 16. Database rewrite rules

Any expression e can be transformed into an equivalent SQL query by repeated application of the rewrite rules given in Figure 16. To give the rewriting process sufficient freedom, we extend the non-terminal s from Figure 14 to give ourselves a working grammar.

```
          s ::= ... | q
```

We write `s[b/x]` for the substitution of b for x in s. Again we abuse the vector notation in the obvious way in order to denote simultaneous pointwise substitution. Note that *min* is a meta language (i.e. compiler) operation that returns the minimum of two integers, and not part of the target language.

Proposition 1. *The database rewrite rules are strongly normalising and confluent.*

Proof. Define the size $|e|$ of an expression e as follows.

$$\begin{aligned}
|\texttt{take}(n,\ s)| &= 1 + |s| \\
|\texttt{drop}(n,\ s)| &= 1 + |s| \\
|\textbf{for}\ (x\ \texttt{<-}\ s1)\ s2| &= 2 + |s1| + |s2| \\
|\textbf{for}\ ((\bar{f} = \bar{x})\ \texttt{<-}\ s1)\ s2| &= 1 + |s1| + |s2| \\
|\textbf{let}\ x = b\ \textbf{in}\ s| &= 1 + |s| \\
|\textbf{where}\ (b)\ s| &= 1 + |s| \\
|\textbf{table}\ t| &= 1 \\
|[(b)]| &= 1
\end{aligned}$$

Each rewrite rule strictly reduces the size of an expression, thus the rules are strongly normalising. The rewrite rules are orthogonal and hence weakly confluent. Confluence follows as a direct consequences of weak confluence and strong normalisation.

5 Statically typed message passing

Concurrency in Links is based on processes that send messages to mailboxes. We here present a tiny core calculus to capture the essence of process and message typing in Links. We let A, B, C, D range over types, s, t, u range over terms, and x range over variables. A type is either the empty type $\mathbf{0}$; unit type $\mathbf{1}$; a process type $P(A)$, for a process that accepts messages of type A; or a function type $A \to^C B$, for a function with argument of type A, result of type B, and that may accept messages of type C when evaluated. Links also supports record types and variant types (using row typing), but as those are standard we don't describe them here. Typically, a process will receive messages belonging to a variant type, and we use the empty variant type (which is isomorphic to the empty type $\mathbf{0}$) to assert that a function does not receive messages.

A typing judgement has the form $\Gamma; C \vdash t : A$, where Γ is a typing context pairing variables with types, C is the type of messages that may be accepted during execution of the term, t is the term, and A is the type of the term.

The typing rules are shown in Figure 17. The rules for variables and the constant of unit type are standard. The type of a lambda abstraction is labelled with the type C of messages that may be received by its body, where the context of the abstraction may accept messages of an unrelated type D; while the function in an application must receive messages of the same type C as the context in which it is applied.

If t is a term of unit type that receives messages of type C, then spawn t returns a fresh process identifier of type $P(C)$, where the context of the spawn may receive messages of an unrelated type D. The term self returns the process identifier of the current process, and so has type $P(C)$ when it appears in a context that accepts messages of type C. If s is a term yielding a process identifier of type $P(D)$ and t is a term of type D, then send s t is a term of unit type that sends the message t to the process identified by s; it evaluates in a context that may receive messages of an unrelated type C. Finally, the term receive returns a value of type C when it is executed in a context that may receive messages of type C.

Links syntax differs slightly from the core calculus. The type $A \to^C B$ is written `A -{C}-> B`, or just `A -> B` when `C` is unspecified. The core expression self is written as a function call `self()`, the core expression send t u is written `t ! u`, and the Links expression

```
receive {
  case p1 -> e1
  ...
  case pn -> en
}
```

corresponds to a switch on the value returned by the receive operator.

We give a formal semantics for the typed message-passing calculus via a translation into $\lambda(\mathsf{fut})$, Niehren et al's concurrent extension of call-by-value λ-calculus [20]. For our purposes we need not consider all of the details of $\lambda(\mathsf{fut})$. We just describe the image of the translation and then rely on previous results [20]. The image of the translation is simply-typed call-by-value λ-calculus extended with the constants of Figure 18.

The expression thread $(\lambda x.u)$ spawns a new process binding its *future* to the variable x in u and also returning this future. A future can be thought of as placeholder for a value that is set to contain a real value once that value becomes known. Futures are central to $\lambda(\mathsf{fut})$, but not to our translation.

The other constants are used for manipulating asynchronous *channels*, which can be used for implementing mailboxes. The constants newChannel, put and get are used for creating a channel, writing to a channel, and reading from a channel. They are not built in to $\lambda(\mathsf{fut})$, but are easily implemented in terms of $\lambda(\mathsf{fut})$-primitives, when $\lambda(\mathsf{fut})$ is extended with product types. The type of channels is $Chan(A)$[2].

We write $\Gamma \vdash^{\mathsf{fut}} u : A$ for the $\lambda(\mathsf{fut})$ typing judgement u has type A in context Γ. The translation on terms and types is given in Figure 19. At the top-level a fresh channel corresponding to the mailbox of the main process is introduced. Thereafter, the translation on terms is annotated with the current channel. Unit and variables are just mapped to themselves. The current mailbox is threaded through functions and applications. The spawn, send and receive constructs are mapped to thread, put and get, and self is simply mapped to the channel that corresponds to the current mailbox. The type translation

[2] In fact $Chan(A)$ is just the type $(A \to unit) \times (unit \to A)$ (i.e. a pair of put and get functions), but that is not relevant to our presentation.

maps unit to unit, process types to channel types, and function types to function types with an extra channel argument.

It is straightforward to prove by induction on typing derivations that the transformation respects typing judgements.

Proposition 2. *The transformation* $[\![\cdot]\!]$ *respects typing judgements.*

$$\Gamma; C \vdash u : A \quad \textit{iff} \quad [\![\Gamma]\!] \vdash^{\mathsf{fut}} \mathsf{let}\ ch^{[\![C]\!]}\ \mathsf{be\ newChannel}\ ()\ \mathsf{in}\ [\![u]\!]^{ch} : A$$

By Proposition 2 and standard results for $\lambda(\mathsf{fut})$ [20], we can be sure that programs that are well-typed in our message-passing calculus "do not go wrong".

6 Issues

Links is still at an early stage of development. There are a number of shortcomings in the current design, which we plan to address. We mention some of them here.

Form abstractions. A flaw in the design of forms is that Links does not support abstraction over form components. As described in section 2.1, form interaction in Links is specified using the attributes `l:name`, which binds the value of a form field to a Links variable, and `l:onsubmit`, which contains Links code to be executed when a form is submitted. Variables bound by `l:name` are in scope within the `l:onsubmit` attribute of the lexically enclosing form. This design has several shortcomings. Firstly, it is not possible to abstract over input elements, since all input elements must be lexically contained within the form to which they belong. Secondly, it is not possible to vary the number of input elements within a form, since each input element is associated with a Links variable, and the number of variables is fixed at compile time. Thirdly, form components are not composable: it is not possible to compose input elements to create new form components which have the same interface as the language-supplied primitives, since `l:name` is only attachable to `input` XML literals.

To illustrate these problems, say we wish to construct a *date* component which we will use to allow the user to enter a number of dates into a form.

```
var date =
  <#>
   month: <input l:name="month"/>
   day:   <input l:name="day"/>
  </#>
```

We would then like to plug instances of *date* into a form created elsewhere in the program:

```
<form l:onsubmit="{e}">
  Arrival:   {date}
  Departure: {date}
</form>
```

$$\frac{}{\Gamma, x : A; C \vdash x : A} \qquad \frac{}{\Gamma; C \vdash () : \mathbf{1}}$$

$$\frac{\Gamma, x : A; C \vdash u : B}{\Gamma; D \vdash \lambda x.u : A \to^{C} B} \qquad \frac{\Gamma; C \vdash s : A \to^{C} B \quad \Gamma; C \vdash t : A}{\Gamma; C \vdash st : B}$$

$$\frac{\Gamma; C \vdash t : \mathbf{1}}{\Gamma; D \vdash \mathsf{spawn}\ t : P(C)} \qquad \frac{}{\Gamma; C \vdash \mathsf{self} : P(C)}$$

$$\frac{\Gamma; C \vdash s : P(D) \quad \Gamma; C \vdash t : D}{\Gamma; C \vdash \mathsf{send}\ s\ t : \mathbf{1}} \qquad \frac{}{\Gamma; C \vdash \mathsf{receive} : C}$$

Fig. 17. Statically typed message passing

$$\mathsf{thread} : (A \to A) \to A$$

$$\begin{aligned}
\mathsf{newChannel} &: \mathit{unit} \to \mathit{Chan}(A) \\
\mathsf{put} &: \mathit{Chan}(A) \to A \to \mathit{unit} \\
\mathsf{get} &: \mathit{Chan}(A) \to \mathit{unit} \to A
\end{aligned}$$

Fig. 18. Constants for the $\lambda(\mathsf{fut})$ translation

$$\llbracket u \rrbracket = \mathsf{let}\ ch\ \mathsf{be}\ \mathsf{newChannel}\ ()\ \mathsf{in}\ \llbracket u \rrbracket^{ch}$$

$$\begin{aligned}
\llbracket () \rrbracket^{ch} &= () & \llbracket \mathsf{spawn}\ u \rrbracket^{ch} &= \mathsf{let}\ ch'\ \mathsf{be}\ \mathsf{newChannel}\ ()\ \mathsf{in} \\
& & &\qquad \mathsf{let}\ \_\ \mathsf{be}\ \mathsf{thread}\ (\lambda\_.\llbracket u \rrbracket^{ch'})\ \mathsf{in}\ ch' \\
\llbracket x \rrbracket^{ch} &= x & \llbracket \mathsf{send}\ s\ t \rrbracket^{ch} &= \mathsf{put}\ \llbracket s \rrbracket^{ch}\ \llbracket t \rrbracket^{ch} \\
\llbracket \lambda x.u \rrbracket^{ch} &= \lambda x.\lambda ch'.\llbracket u \rrbracket^{ch'} & \llbracket \mathsf{receive} \rrbracket^{ch} &= \mathsf{get}\ ch \\
\llbracket st \rrbracket^{ch} &= \llbracket s \rrbracket^{ch}\ \llbracket t \rrbracket^{ch}\ ch & \llbracket \mathsf{self} \rrbracket^{ch} &= ch
\end{aligned}$$

$$\llbracket \mathbf{1} \rrbracket = \mathbf{1} \qquad \llbracket P(A) \rrbracket = \mathit{Chan}(\llbracket A \rrbracket) \qquad \llbracket A \to^{C} B \rrbracket = \llbracket A \rrbracket \to \llbracket P(C) \rrbracket \to \llbracket B \rrbracket$$

Fig. 19. Translation into $\lambda(\mathsf{fut})$

Unfortunately, Links provides no way to refer within the `l:onsubmit` attribute to the values entered into the date fields, since the input elements are not lexically enclosed within the form. The only way to write the program is to textually include the date code within the form, then refer to the individual input elements of each month and day within *e*. This violates abstraction in two ways: the component must be defined (twice!) at the point of use, and the individual elements of the component cannot be concealed from the client.

If we try to construct a form with a number of fields determined at runtime then we run into further difficulties:

```
fun (n) {
  <form l:onsubmit="{e}">{
   for (var i <- range(n))
     Field {i}: <input l:name="x"/>
  }</form>
}
```

This code is also problematic: there is no way to refer to the different input fields generated in the loop, since all have the same name! Such constructs are consequently disallowed, and Links provides no way to construct forms with dynamically-varying numbers of fields.

The `l:name` mechanism provides a limited (but clean) way to refer to DOM nodes that are `<input>` elements. A related problem is that there is currently no other way of statically referring to a DOM node, which means we have to resort to calling the function *getNodeById* as illustrated in the draggable lists example.

Other systems also suffer from these problems to a greater or lesser degree. In PLT Scheme [15], composing new forms from old or making forms with a varying number of fields is possible, but not straightforward. In WASH [31] and iData [23], building forms that return new types requires declaring instances for those types in particular type classes, and in WASH a special type constructor needs to be used for forms with a varying number of fields. It was surprising for us to realize that composability of forms has received relatively little attention.

We are currently working on a design for light-weight modular form components, which uses a uniform framework to support form abstraction, varying numbers of form fields, and a mechanism that allows DOM nodes to be statically bound.

Synchronous messages. As far as we are aware, our mailbox typing scheme for Links is novel, though the translation into $\lambda(\mathsf{fut})$ shows that it can be simulated using typed concurrent languages with support for typed channels.

Links message passing is asynchronous, but often one wants synchronous message passing, where one sends a message and waits for a result. An example where synchronous message passing is needed is the extension of the draggable lists example to support reading the content of draggable lists.[3]

Although Links does not directly support synchronous message passing, it can be simulated in the same way as in Erlang. In order to send a synchronous message from process P to process Q the following sequence of events must occur.

[3] See `http://groups.inf.ed.ac.uk/links/examples/`.

– P sends a message (M, P) to Q
– P blocks waiting for a reply $\mathit{Reply}(N)$
– on receiving the message (M, P) the process Q sends a reply message $\mathit{Reply}(N)$ to P
– P receives the reply message $\mathit{Reply}(N)$

However, Erlang is untyped; in Links, the simulation of synchronous messages has the unfortunate side-effect of polluting the type of process P's mailbox, to include a different constructor for each distinct type of reply message associated with synchronous messages sent by P.

Type pollution of this kind makes it hard to structure programs cleanly. We are considering moving to a different model of concurrency that makes it easier to support typed synchronous message passing, such as the Join calculus [10], or even removing explicit concurrency from the language altogether and replacing it with something like functional reactive programming [33].

Database abstraction. As we showed in Section 4, we can guarantee to compile a useful fragment of Links into SQL. However, this fragment still lacks many important features. We do not yet deal with aggregate functions such as `count`, `sum`, and `average`, we only support `order by` attached to a single loop iterator (rather than a nest of loop iterators). One can express the equivalent of a `group by` construct as a nested loop, but we do not yet translate code into `group by` when appropriate, nor yet support any other form of nested queries.

We also do not yet provide adequate support for abstracting over a query. For instance, suppose we try to abstract over the condition in the dictionary suggest example. We can amend the *completions* function to take a condition encoded as a function from unit to bool in place of the prefix.

```
fun completions(cond) server {
 if (s == "") [] else {
  take(10, for (var def <-- defsTable)
           where (cond) orderby (def.word)
            [def])
 }
}
```

Only once the condition is known is it possible to compile this to efficient SQL. The current compiler produces spectacularly inefficient SQL, which returns the entire dictionary, leaving the Links interpreter to then filter according to the condition and then take the first 10 elements of the filtered list. To produce adequately efficient code, it is necessary to inline calls to *completions* with the given predicate. But our compiler does not yet perform inlining, and in general inlining can be difficult in the presence of separate compilation.

Some of the problems outlined above are easy. For instance, we should be able to support aggregates in the same style that we support `take` and `drop`; and it is straightforward to extend our techniques to support `orderby` on nested loops. However, other problems are more challenging; in particular, it is not immediately clear how to extend the fragment described in Section 4 to support abstraction over predicates in a useful way.

Microsoft's Linq addresses all these issues, but does so by (a) requiring the user to write code that closely resembles SQL and (b) a subtle use of the type system to distinguish between a procedure that returns a value and a procedure that returns code that generates that value. It remains an open research problem to determine to what extent one can support efficient compilation into SQL without introducing separate syntax and semantics for queries as is done in Linq.

7 Conclusion

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 and draggable lists. Our programming language compiles to JavaScript to run in the client, and SQL to run on the database.

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. 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.
2. J. Armstrong. Concurrency oriented programming in Erlang. Invited talk, FFG 2003.
3. V. Balat. Ocsigen: typing web interaction with objective Caml *Proceedings of the 2006 workshop on ML*, Portland, Oregon, September 2006.
4. N. Benton, L. Cardelli, C. Fournet. Modern concurrency abstractions for C♯. *TOPLAS*, 26(5), 2004.
5. N. Benton, A. Kennedy, and C. Russo. Adventures in interoperability: the SML.NET experience. *PPDP*, 2004.
6. G. Bierman, E. Meijer, W. Schulte. Programming with rectangles, triangles, and circles. *XML Conference*, 2003.
7. P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *TCS*, 149(1), 1995.
8. R. Burstall, D. MacQueen, and D. Sannella. Hope: An experimental applicative language. *Lisp Conference*, 1980.
9. 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.
10. C. Fournet, G. Gonthier. The Join Calculus: a language for distributed mobile programming. *Applied Semantics*, LNCS 2395, 2002.

11. J. Garret. Ajax: a new approach to web applications. 2005.
12. P. Graham. “Method for client-server communications through a minimal interface.” United States Patent no. 6,205,469. March 20, 2001.
13. P. Graham. Beating the averages. 2001.
14. P. Graunke, R. B. Findler, S. Krishnamurthi, M. Felleisen. Automatically restructuring programs for the web. *ASE*, 2001.
15. P. Graunke, S. Krishnamurthi, S. van der Hoeven, M. Felleisen. Programming the web with high-level programming languages. *ESOP*, 2001.
16. V. Gapeyev, M. Levin, B. Pierce, A. Schmitt. The Xtatic experience. *PLAN-X*, 2005.
17. `labs.google.com/suggest`
18. Microsoft Corporation. *DLinq: .NET Language Integrated Query for Relational Data* September 2005
19. A. Mller, M. Schwartzbach. The design space of type checkers for XML transformation languages. *ICDT*, LNCS 3363, 2005.
20. J. Niehren, J. Schwinghammer, G. Smolka. A Concurrent Lambda Calculus with Futures. *TCS*, 364(3), 2006.
21. G. Narra. ObjectGraph Dictionary.
`http://www.objectgraph.com/dictionary/how.html`
22. M. Odersky *et al*. An overview of the Scala programming language. Technical report, EPFL Lausanne, 2004.
23. R. Plasmeijer and P. Achten. iData For The World Wide Web: Programming Interconnected Web Forms. *FLOPS*, 2006.
24. 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.
25. C. Queinnec. Continuations to program web servers. *ICFP*, 2000.
26. Christian Queinnec *Inverting back the inversion of control or, continuations versus page-centric programming*, SIGPLAN Not., 2003
27. J. Reynolds. *Definitional interpreters for higher-order programming languages*. ACM ’72: Proceedings of the ACM annual conference, 1972.
28. Ruby on Rails.
http://www.rubyonrails.org/
29. M. Serrano, E. Gallesio, and F. Loitsch. HOP, a language for programming the Web 2.0. *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, October 2006.
30. D. Syme. F♯ web page.
`research.microsoft.com/projects/ilx/fsharp.aspx`
31. P. Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. *PADL*, 2002.
32. P. Van Roy. Convergence in language design: a case of lightning striking four times in the same place. *FLOPS*, 2006.
33. Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI ’00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver, British Columbia, Canada, 2000.
34. B. Wiederman and W. Cook. Extracting queries by static analysis of transparent persistence. *POPL*, 2007.
35. L. Wong. Kleisli, a functional query system. *JFP*, 10(1), 2000.
36. XML Query and XSL Working Groups. XQuery 1.0: An XML Query Language, W3C Working Draft, 2005.