Using Links to prototype a Database Wiki

James Cheney, Sam Lindley
University of Edinburgh
jcheney@inf.ed.ac.uk,
Sam.Lindley@ed.ac.uk

Heiko Müller
Tasmanian ICT Centre
CSIRO
Hobart, Australia
heiko.mueller@csiro.au

ABSTRACT

Both relational databases and wikis have strengths that make them attractive for use in collaborative applications. In the last decade, database-backed Web applications have been used extensively to develop valuable shared biological references called *curated databases*. Databases offer many advantages such as scalability, query optimization and concurrency control, but are not easy to use and lack other features needed for collaboration. Wikis have become very popular for early-stage biocuration projects because they are easy to use, encourage sharing and collaboration, and provide built-in support for archiving, history-tracking and annotation. However, curation projects often outgrow the limited capabilities of wikis for structuring and efficiently querying data at scale, necessitating a painful phase transition to a database-backed Web application. We perceive a need for a new class of general-purpose system, which we call a *Database Wiki*, that combines flexible wiki-like support for collaboration with robust database-like capabilities for structuring and querying data. This paper presents DBWiki, a design prototype for such a system written in the Web programming language Links. We present the architecture, typical use, and wiki markup language design for DBWiki and discuss features of Links that provided unique advantages for rapid Web/database application prototyping.

Keywords

curated databases, web programming, Links, rapid prototyping

1. INTRODUCTION

Relational databases underlie many of the commercial systems and Web applications in use today, and play a central role in large-scale scientific data management, especially in biological sciences. However, direct access to databases has largely remained the preserve of professional programmers and database administrators. Relational databases remain hard to use, in part because their high-level interfaces — usually variants of SQL — have remained essentially unchanged since their introduction in the 1970s and remain difficult to reconcile with other programming models. Moreover, relational databases lack native support for features such as versioning, annotation, and provenance, which are especially important for scientific databases [7]. Many of these features can be added to existing systems in ad hoc ways, but this can be expensive.

Wikis are user-editable Web sites that have grown very popular in the last decade. A prime example is Wikipedia, which is displacing standard print reference works. Wikis allow users to edit their content almost as casually as they search or browse. Wikis support collaboration and transparency by recording detailed change histories and allowing space for discussion. Because they are free and relatively simple to set up and configure, wikis are becoming popular for nascent biological database projects: for example, the Gene Wiki Portal[1] lists over 15 biological wiki projects, and a wiki was used to coordinate scientific response to the swine flu outbreak in early 2009[2].

There is a basic tension between structure and flexibility. Systems such as MediaWiki (used by Wikipedia and many biological database projects) employ relational database technology internally to provide efficient and robust concurrent access to many users, but the data managed by these systems is still (essentially) text or HTML. Many kinds of data stored in wikis actually have regular structure, but are stored as HTML tables or lists. Wikis encourage writing unstructured text, which can be searched using standard information-retrieval techniques, but this approach does not scale to the data sizes and complex queries that are needed, for example, in curated biological databases.

Biocuration projects are expensive, in part because the data is manually selected and edited by experts. It is important to record not only the data (including past versions) but also *provenance* metadata about the creation and history of the data and *annotations* describing opinions or discussion of the data [7]. This extra functionality is often neglected, or reimplemented in different systems in ad hoc or unreliable ways. We believe that the needs of database curation projects could be met more reliably and cost-effectively by developing new general-purpose systems that combine the advantages of databases and wikis. We call such systems *Database Wikis*. Much of the basic research on curated databases needed to implement database wikis, such as archiving, citation, provenance, and annotation management, has already been conducted [8, 22, 5, 4, 7]. However, there is no single system that draws these techniques together.

In this paper, we present a Database Wiki design prototype called DBWiki, implemented in Links [12]. The contributions of this paper include lessons learned about both the design of Links and Web programming languages, and about the tradeoffs involved in incorporating database query and update features into a wiki markdown

[1] http://en.wikipedia.org/wiki/Portal:Gene_Wiki/Other_Wikis

[2] http://tree.bio.ed.ac.uk/groups/influenza/

language. We are incorporating these lessons into Links and into a production Database Wiki system.

Outline. Section 2 presents the design of the DBWiki system at a high level, with examples of use, while Section 3 briefly introduces Links. Section 4 discusses salient aspects of the implementation in Links. Section 5 discusses the advantages and disadvantages of using Links compared to a Java reimplementation. Section 6 discusses related work, and Section 7 concludes.

2. DESIGN OVERVIEW AND EXAMPLES

The Links Database Wiki prototype is essentially a basic wiki combined with a semi-structured database (or *data tree*) with built-in archiving, annotation and provenance tracking. The data tree is an unordered, edge-labeled deterministic tree. Determinism means that each path addresses exactly one subtree. The motivation for this is to ensure that paths can be used as unique identifiers for subtrees. This is a flexible and generic data model that has been employed in previous work on different facets of curated databases [8, 5, 22]; although we believe a more sophisticated model is needed for real applications, we adopted this simple data model to facilitate rapid prototyping.

DBWiki also includes "wiki pages", which are written in a simple concrete syntax based on that of common wikis. We adapted a wiki syntax parser developed in the SEWiki application for SELinks (and made available by the SEWiki developers [15]). Wiki pages have the following abstract syntax:

$$
\begin{array}{rcl}
p & ::= & /\ell \mid /\$x; \mid p\ p' \\
A & ::= & \mathsf{string} \mid [A] \mid \{\ell_1 : A_1, \ldots, \ell_k : A_k\} \\
 & \mid & \mathsf{editable}(A) \mid \mathsf{link} \\
tag & ::= & \mathsf{h1} \mid \mathsf{h2} \mid \mathsf{em} \mid \mathsf{b} \mid \mathsf{blockquote} \mid \mathsf{br} \mid \cdots \\
P & ::= & s \mid tag[s] \mid [[Name]] \mid ![s]\ (url) \mid [s]\ (url) \mid \epsilon \mid P\ P' \\
 & \mid & ?p? \mid ?\langle A \rangle p? \mid !p! \mid [[Name(x_1 = s_1, \ldots, x_n = s_n)]]
\end{array}
$$

Paths p are essentially just sequences of labels or variables $\$x;$. Types A include base type string, record types $\{\ell_1 : A_1, \ldots, \ell_k : A_k\}$, list types $[A]$ and additional types used for rendering, discussed below. HTML tags include all of the standard tags used in Wiki-like markup languages.

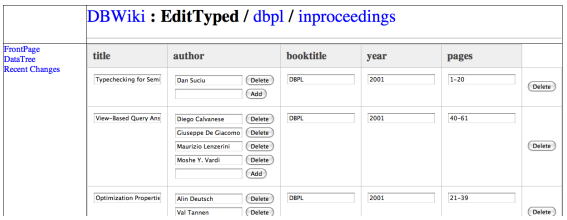
Pages P are, essentially, sequences consisting of strings interspersed with markup and some special constructs that support interaction with the data component of the wiki. Here, ϵ stands for the empty page, $P\ P'$ stands for page concatenation, and $tag[s]$ stands for an HTML element with content string s. The constructs $[s]\ (url)$ and $![s]\ (url)$ provide links and image links to external resources addressed by URLs respectively, while the construct $[[Name]]$ is a "wiki link" pointing to a page named $Name$. So far, these constructs are standard elements of wiki page syntax; in addition, we provide syntax for embedding *views*, *typed views* and *data links* to the data in the tree. We also support templates (that is, pages with *parameters* and *parameterized wiki links*).

A view, or embedded query, is written as $?p?$, and by default this renders the data at path p in a generic way. This is not always readable, so in addition we allow views to take an optional *rendering type* A, which describes the expected form of the data at p. Thus, a typed view $?\langle A \rangle p?$ instructs the system to extract the data at path p and render it as if it were of type A. Rendering types include base type, record and set constructors as well as a special *editable* type $\mathsf{editable}(A)$ and a *link* type link that indicates a string that should be rendered as a clickable URL.

Rendering a path query as a list of records is treated as a special case: as lists of records often correspond to tables, they are presented as HTML tables with a single header row for the field names.

(a)

(b)

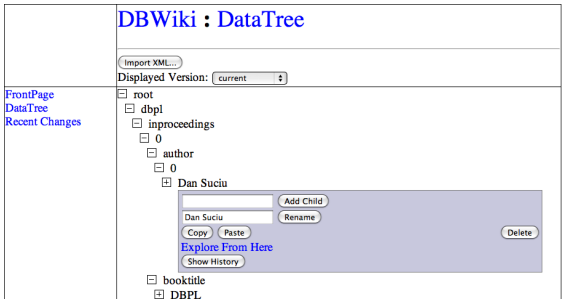


(c)

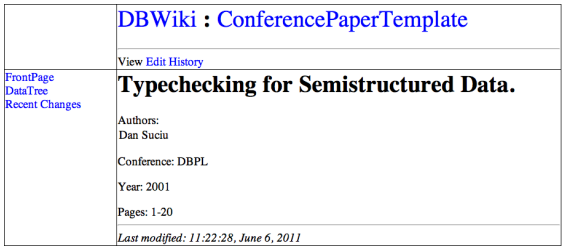
```
# Some conference papers #

An editable table:
?<editable([{title:string,author:[string],
      booktitle:string,year:string,
      pages:string}])>/dbpl/inproceedings?
```

(d)



(e)



(f)

```
# ?<string>/dbpl/inproceedings/$id;/title? #

Authors:
?<[string]>/dbpl/inproceedings/$id;/author?

Conference:
?<string>/dbpl/inproceedings/$id;/booktitle?

Year:
?<string>/dbpl/inproceedings/$id;/year?

Pages:
?<string>/dbpl/inproceedings/$id;/pages?
```

Figure 1: DBWiki screenshots and source code examples. (a) A Wiki page rendering the data at `/dbpl/inproceedings` as a table. (b) An editable version of the table. (c) Source for (a,b). (d) The data tree editor. (e) A template page. (f) Source for (e).

Figure 1(a) shows the rendering of some DBLP bibliographic data about DBPL 2001, stored at path `/dbpl/inproceedings`, as a table. It also illustrates the use of the `link` type. Wrapping a type in `editable` causes an edit hyperlink to be inserted in the rendering of the corresponding binding. Clicking on this link takes the user to the form-based editing interface for that binding. For example, Figure 1(b) shows the form generated for 1(a); Figure 1(c) shows the markdown source for the editable table.

A data link $!p!$ simply provides a link to the data at path p. Following this link yields a tree viewer/editor rooted at p. For example, Figure 1(d) shows a (partial) tree editor view of the subtree at path `/dbpl`. The tree editor supports arbitrary changes to the data tree via a Web browser, including inserting, deleting, renaming edges and copying and pasting subtrees. XML data can also be parsed and imported into the tree.

Path steps can be *parameter references* `$x;`. These refer to additional optional arguments (passed in as CGI parameters in the URL), making it possible to write *templates* that can be filled in with data from different parts of the tree. Likewise, wiki links can be written with explicit parameters $[[Name(x_1 = s_1, \ldots, x_n = s_n)]]$. These links are translated to URLs which link to the appropriate template page and fill in the CGI arguments with the specified strings. This makes it possible to write "index" pages that link to a number of different instantiations of a template. Figure 1(e) illustrates an example of a page rendered using a template to fill in data from a record in the `/dbpl/inproceedings` subtree, and Figure 1(f) shows the template source.

We are developing a formal semantics of page rendering and the behavior of updates (including versioning and provenance behavior), extending the approach described in prior work [5]. This semantics will be needed to ensure correctness and type-safety in the presence of concurrent access (a topic for future work).

3. LINKS OVERVIEW

Links is a functional programming language that allows writing a Web application as a single program and splitting it into a server-side program, client-side JavaScript and HTML, and SQL queries against relational databases [12, 14, 11].

Basic concepts. Links is a functional, higher-order, call-by-value, impure, typed language. Its expression syntax is loosely based on JavaScript, while its type syntax is loosely based on that of Haskell. The type language supports ordinary polymorphism, polymorphic variants, row types, and effect polymorphism (which is currently used to support concurrency and database queries). In addition to standard functional programming constructs such as pattern matching (`switch/case`), Links provides list comprehensions (`for`) and XML literals with antiquoting, similar to XQuery or Scala. The latter allows code and HTML to be freely mixed, generalizing the functionality of mainstream web scripting languages such as PHP and JSP.

Execution model. A Links program running in *web mode* is split, by the Links runtime, into two parts: an HTML page that is sent to the client when the program is first invoked (via an HTTP request), and a server-side component (currently implemented as an interpreter) which responds to further XML HTTP Requests (XHRs) from the client. The client-side code may contain embedded JavaScript that responds to user interaction events and makes XHR calls back to the server. Links functions can be explicitly declared `client` or `server`; in the absence of an explicit annotation, most code can run in either domain (exceptions include queries and concurrency, as highlighted below).

Like a number of other Web programming languages, Links programs support arbitrary transfer of control flow between client and server by serializing server-side continuations. Such continuations can appear in URLs, HTTP POST requests and values returned from XML HTTP Requests (for implementing server→client calls). Serializing continuations has limitations: it can yield URLs with large, unreadable CGI arguments, and many Web servers (including Apache) place an upper limit of around 4KB on URL length. In DBWiki we wanted to keep URLs readable and stable over time in order to allow external sites or searches to link to particular wiki pages, as most wikis do, so we deliberately avoided the use of serialized continuations in URLs. Links also provides advanced features for creating Web forms and handling the resulting HTTP POST requests, through an interface called *formlets* [13]. However, currently formlets are targeted at classical synchronous Web 1.0-style forms, and since most forms in DBWiki are dynamic we did not make heavy use of formlets.

Types and effects. The Links type system explicitly captures certain *effects* in the source language. The idea of an effect type systems is best understood in terms of functions. In a plain type system, each function has an argument type (or collection of argument types for multi-argument functions) and a return type. In an effect type system, each function also has an *effect type*. Whereas the argument and return types capture the shape of inputs and outputs, the effect type captures what kind of operations the function is allowed to perform. For instance, Java supports a kind of effect type in the form of exception specifications that indicate which exceptions a method is allowed to raise.

Currently only two effects are captured in Links, to deal with concurrency and database programming. The first effect `hear` : A is used for concurrency. It indicates that the current process can *hear* messages of type A. The second effect `wild` indicates code that *cannot* be compiled to the database, such as code involving general recursion or concurrency.

Functions in Links take n-tuple arguments. Function types

```
(A1, ..., An) {E}-> B
```

have n argument types `A1`, ..., `An`, an effect `E`, which is written as an annotation on the arrow, and a return type `B`. In common with Blume et al. [3], *row types* are used to encode effects. In order to make types (and errors) more readable to users, effects are often hidden or abbreviated with syntactic sugar. For the purposes of this paper, all the reader needs to know is that `->` indicates a *tame* (i.e., non-wild) function that can be used inside a query, whereas `~>` indicates a wild function that cannot be used inside a query, and `~e~>` indicates a wild function with polymorphic effect variable `e`.

Language-integrated query. Links provides high-level support for connecting to relational databases (including MySQL and PostgreSQL), extracting data using queries, and updating data. Links provides comprehension syntax for queries. Comprehensions suffice to express many common database queries [9], and using comprehension syntax instead of explicit recursion or `fold` operations over lists makes it possible to recognize many query idioms automatically. Links will also attempt to translate nested `for`-loops to single queries. For example:

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

turns into (modulo renaming):

```
SELECT x.a, y.d FROM foo x, bar y WHERE x.a = y.c
```

Note that this query could be written in a number of other ways, e.g. using SQL's `JOIN` keyword; Links generates queries in a `SELECT-FROM-WHERE` form and does not try to identify idiomatic SQL, instead relying on the relational optimizer.

Links provides a keyword `query` that asserts that a block of code translates to at most one query. The effect-type system statically enforces this constraint by disallowing code that cannot be translated to a query inside a query block. At run time, Links uses a normalization procedure (generalizing an approach taken in Wong's Kleisli system [27]) to convert query blocks to normal forms that correspond to isomorphic SQL queries. Operationally, this is a form of run-time code generation: actually, run-time *query* generation. Moreover, query blocks can refer to higher-order functions, as long as they are tame (that is, only use features that can be performed by SQL queries). Violations are reported as type errors if the `query` block contains computations that cannot be expressed using queries, such as unbounded recursion or concurrency. Cooper [11] presents the initial design of this feature of Links (including both the type/effect system and rewrite system with proof of strong normalization), which inspired the current implementation.

An important property of the normalization procedure is *predictability*. It produces idiomatic SQL. If the programmer writes a comprehension that is in normal form, then it is translated to the isomorphic SQL query. If the programmer abstracts over part of a where clause then the query is the same as the query one would obtain by inlining the abstraction in the where clause.

The form `for (x <-- t) e` is syntactic sugar for

```
query {for (x <- asList(t)) e},
```

where the built-in function `asList` presents a table as a list of rows.

4. IMPLEMENTATION OF DBWIKI

Data model. We implement the nested record structure in Links as a list of bindings, where a binding maps a string label to another record.

```
# records as lists of bindings
typename Binding(a) = mu b.(String, ([b], a));
typename Record(a) = [Binding(a)];
```

(These `typename` declarations introduce type aliases. The `mu` form introduces an equi-recursive type [23, ch. 20–21].) We also use a standard zipper data structure for navigating through the nested record structure [19]. For convenience, we parameterize these types by an annotation type `a`, such as a unique identifier to link a binding to its location in a user interface, or a flag to indicate whether it has been saved to the database. Often, these in-memory data structures (or their HTML/JavaScript presentations) essentially correspond to *updatable views* of the underlying database. We use the identifiers to translate updates to these views to updates to the underlying database. This is relatively straightforward because the class of views is relatively small; bidirectional programming techniques could be applied to handle richer classes of views, and this is an interesting direction for future work. In particular, Foster et al. [16] consider views on a tree-structured data model very similar to ours.

Relational back-end. The persistent data of a DBWiki instance is stored in tables for bindings, binding names, time intervals, wiki pages, and provenance. The Links database and table declarations for DBWiki are shown in Figure 2. For the moment, note that it declares a database `dbwiki` and a table in the `dbwiki` database whose name is `bindings`, with columns `id`, `row` and `child` of type `Int`, and `name` of type `String`. The constraint `id readonly` expresses that the `id` column is read-only. The `id` column is the primary key for bindings. Its presence allows us to change the name of a binding (and preserve its history) without deleting and inserting an entire subtree. The `row` field identifies the record in which this binding appears. The `name` field is the name of the binding, and the `child` field identifies the record containing its children.

We use `query` blocks to construct queries for looking up a path in the tree. First consider a naive implementation that generates multiple SQL queries, shown in Figure 3. The function `lookupPath0` takes a path consisting of a list of strings, and returns the binding id and row id for the binding at the end of the path. The auxiliary function `lookup` recursively traverses the supplied path, generating a query to look up the next binding at each step. We assume that paths are unique. If `Nothing` is returned, this indicates either that the path is not present, or that it is not unique.

It is often more efficient to evaluate a single query per path rather than one query per path step. As a first attempt, we could simply wrap the call to `lookup` in a `query` block, to try to force the result to be obtained by a query. This leads to a type error, because `lookup` is recursive and so the effect analysis infers that calls to it cannot be compiled to SQL. The solution is to move the recursion out of the query itself. We use recursion to assemble the query, but no recursion appears in the resulting query, as shown in Figure 4. Now the auxiliary `lookup` function takes the path and also the query constructed so far. The query is built up as a function from the root binding to a list of bindings (which we expect to be a singleton). The `lookup` function is recursive, but the result it returns is not. The query block surrounding the call to `q` does not give a type error, so we can now be sure that it will compile to at most one query.

The encoding of query components as functions is important for two reasons. First, it allows the query components to be composed before the query is actually run. Second, it allows inner components to depend on values computed from outer components. The pattern we use may be familiar to Haskell programmers, where one monad might be used to construct a computation which is subsequently run in a different monad. A difference with Haskell is that one can take advantage of the monad structure to represent the components using monad computations directly rather than as functions.

Archiving. Following the XArch archiving system of Buneman et al. [8, 22], all versions of the data are stored, using time-interval annotations in an `intervals` table. Each interval provides a start and end time during which the associated binding is active. As a convenience, the value 0 is used to indicate that the end of an interval is open, that is, the binding in question is present now. We also track previous names for bindings in table `names`, in order to support efficient renaming (not considered in XArch [8, 22]). We only include the start of the interval in which a binding had a particular name. The end is given by the start of the interval in which the binding next changes its name, or 0 if there are no subsequent name changes. The `name` field of the `bindings` table also contains (redundantly, as a convenience) the current name of the binding.

To further illustrate query integration in Links, we show how to implement looking up a path at a particular time. First, we define a function `nameAt` which returns the name of a binding at a given

```
var db         = "dbwiki";
var bindings   = table "bindings" with (id : Int, row : Int, name : String, child : Int)
                  where id readonly from db;
var intervals  = table "intervals" with (id : Int, binding : Int, start : Int, end : Int)
                 where id readonly from db;
var names      = table "names" with (id : Int, binding : Int, name : String, start : Int)
                 where id readonly from db;
var provenance = table "provenance" with (id : Int, fromtime : Int, frombinding : Int,
                                          totime : Int, tobinding : Int) where id readonly from db;
var pages      = table "pages" with (id:Int, name:String, content:String, timestamp:Int)
                 where id readonly from db;
```

Figure 2: Relational database table declarations for DBWiki

```
sig lookupPath0 : ([String]) ~> Maybe(BindingInfo)
fun lookupPath0(path) server {
  fun lookup(path,
             (binding=_, row=row) as bindInfo) {
    switch (path) {
      case []          -> Just(bindInfo)
      case label::path ->
        switch (for (b <-- bindings)
                where (b.row == row
                       && b.name == label)
                  [(binding=b.id, row=b.child)]) {
          case [bindInfo] -> lookup(path, bindInfo)
          case _          -> Nothing
        }
    }
  }
  lookup(path, (binding=0, row=0))
}
```

Figure 3: Implementation of `lookupPath0` generating multiple queries.

```
sig lookupPath : ([String]) ~> Maybe(BindingInfo)
fun lookupPath(path) server {
  fun lookup(path, q) {
    switch (path) {
      case []          -> q
      case label::path ->
        lookup(path,
               fun ((binding=_, row=row)) {
                 for (b <-- bindings)
                 where (b.row == row
                        && b.name == label)
                   q((binding=b.id, row=b.child))
               })
    }
  }
  var q = lookup(reverse(path),
                 fun (bindInfo) {[(bindInfo)]});
  switch (query {q((binding=0, row=0))}) {
    case [v] -> Just(v)
    case _   -> Nothing
  }
}
```

Figure 4: Implementation of `lookupPath` generating a single query.

time, shown in Figure 5. Notice that the arrow type in the function signature is tame (`~>`), so it can be used inside other queries. Now, we can define the function `lookupPathAt`, which looks up the binding at the end of a path at a specified time, and generates a single SQL query. The only difference compared to `lookupPath` is that the recursive argument to `lookup` uses `nameAt` instead of a name test `b.name == label`. The implementation of `nameAt` is rather more complicated than one might hope due to limitations in the current version of Links. In particular, Links does not currently support aggregation (for example, maximum) inside queries. This is not a fundamental limitation, though; Ferry [18] supports nested data, grouping and aggregation, and these features are being incorporated into Links [25].

Annotations and Provenance. The archiving data immediately provides us with some provenance information. It tells us when a binding was created or deleted, and when a binding had its name changed. In addition we support a copy-and-paste operation, which allows a binding (including all of its descendants) to be copied elsewhere in the database. In this case, we record where it (and the copies of its descendants) came from in a separate `provenance` table. We implement annotations on top of the underlying data model. To annotate a binding, we simply attach a special child binding with the label `_annotation`, and any children of this node are deemed to be annotations of the original binding. This provides history and provenance tracking for annotations at no extra effort.

Concurrency and fresh name generation. Links implements an Erlang-style mailbox concurrency model [1]. In web mode, concurrency takes place only on the client. This model supports a standard idiom for handling user interface (UI) events: we spawn an auxiliary handler process for each independent UI component that waits for messages dispatched to the component. UI events are dispatched by a special process spawned by the Links run-time. In order to ensure that the UI stays responsive, the dispatcher typically just sends a message to the auxiliary handler process.

A UI component handler typically maintains component state that is updated each time it receives a message. Thus one might view the handler process as the *model* component of a "model-view-controller" architecture. Though the events are fired, and messages are sent, asynchronously, the messages are handled synchronously, ensuring that the model always remains in a consistent

```
sig nameAt : (Int, Int) -> [(name:String)]
fun nameAt(binding, t) {
  fun p(n, f) {
    n.binding == binding && f(n) &&
    empty (for (m <-- names)
           where (m.binding == binding
                  && n.start < m.start && f(m))
             [(id=m.binding)])
  }
  for (n <-- names)
  where ((t == 0 && p(n, fun (x) {true})) ||
         (t <> 0 && p(n, fun (x) {x.start < t})))
    [(name=n.name)]
}
```

Figure 5: Function `nameAt` returning a binding name at a particular point in time

state. This approach is a common idiom for the Erlang-style concurrency model adopted by Links.

In several places we need to generate fresh names (e.g. for dynamically generated UI components). An obvious way to implement this is to use mutable state. Links does not provide direct support for mutable references.

Fresh name generation (and, in principle, arbitrary stateful references) can also be implemented safely and systematically using concurrency, and this is the approach taken in DBWiki. Although Links processes are in principle extremely light-weight (just like Erlang processes [1]), in practice there can be a significant penalty for a context switch on the client. This is because context switching uses JavaScript's `setTimeout` function, which is a well-known source of performance problems. (On the server, context switches are essentially free.) We tried this approach for generating fresh names, and ran into problems when generating hundreds of names at once. To fix this we support generating an arbitrary number of names in one go, amortizing the high per-switch cost.

A more subtle issue arose due to effect typing. The issue is really a limitation of the Hindley-Milner type system. Suppose we create a new name supply:

```
sig nameSupply : () ~> (freshId:() ~> Int)
var gen = nameSupply();
```

Now, `gen` is assigned the type `(freshId:() {E}~> Int)` for some fixed effect type `E`. This means we cannot call `gen.freshId` from different effect contexts; in particular we cannot call it from processes with different mailbox types. But, we use the same name generator in the main process during initialisation and in the tree-view handler process. In previous versions of Links, circumventing this problem required an ugly hack. The current version of Links supports first-class (arbitrary rank) type and effect polymorphism, which allows us to assign the more general type

```
freshId:forall e.() ~e~> Int
```

to `gen`, permitting reuse with arbitrary mailbox types.

5. DISCUSSION

Developing the DBWiki prototype revealed both strengths and weaknesses of Links. Links' declarative approach to language integrated query has many benefits. Like other approaches such as LINQ [21] or Ur/Web [10], Links prevents SQL injection by automatically escaping parameters, but unlike any other language of which we are aware, Links also allows queries to be composed using higher-order abstraction mechanisms, mediated by an effect system. We believe that the ability to seamlessly modularize queries using the native Links abstraction mechanisms is powerful and helpful; we leveraged this extensively for `lookupPath` and also for checking database integrity constraints, such as checking that no two intervals for the same binding overlap.

On the other hand, the DBWiki prototype has also pushed the limits of the current Links implementation. DBWiki is one of the largest Links programs written so far, at approximately 4500 lines of code. Developing DBWiki forced us to address some performance issues that had been lurking in the Links interpreter, and strongly motivates developing a compiler for full Links (pending further research on how to compile code that performs run-time query generation). The main obstacles to working entirely in Links at the moment are performance and functionality: there are some common tasks (such as parsing large wiki pages) where Links' performance is not competitive, and others (such as creating new tables and initializing the database) that Links at present cannot perform.

Links is not yet ready for use in development of a production Database Wiki system, but the Links prototype has helped significantly in clarifying design issues. We believe that the design of the user interface is especially important for this kind of system, especially the "markdown" language used to query, update, or visualize structured data from within wiki pages. User interfaces and domain-specific language designs are challenging to evaluate, and can benefit significantly from iterative design based on lightweight prototypes to facilitate iteration and obtain early feedback from potential users. We believe that using Links for rapid prototyping has helped us explore the design space more quickly than would have been possible in a production system.

We are now focusing on an alternative Database Wiki implementation in Java [6], initially due to improved performance and tool support and to make it easier for others (e.g. MSc students) to connect to existing Java libraries and web services. (For example, we currently have one student working on connecting Google Maps and Charts APIs to DBWiki data.) A comparison of the code size of the systems is revealing: the Java version includes over 30,000 lines of code spread across hundreds of source files. The systems provide complementary functionality, and this comparison does not account for features present in the Java version not present in the Links prototype or vice versa. However, if we compare just the code for generating queries from paths, the Java version takes over 100 lines of code, including explicit parameter substitution to prevent SQL injection attacks, whereas the Links version takes only 20 lines of code.

Of course, number of lines of code is not an exact measure of code complexity or maintenance cost. Our experience extending the Java version and supervizing student projects based on it has been mixed. On the one hand, using Java gives us access to more libraries and tools, as well as guaranteeing portability. On the other hand, developing, maintaining and extending the Java version is much more labor-intensive, in part because of the sheer size of the code and in part because of the fact that the Java version performs both dynamic query generation and dynamic HTML generation by composing raw strings using JDBC. This illustrates that a declarative high-level language for Web and database programming can offer significant advantages for rapid prototyping, even in the absence of an optimizing compiler or extensive libraries. However, further development effort needs to be invested, and some research problems may need to be solved, to make the Links approach an unambiguous win for developing a production system compared to a conventional approach.

6. RELATED AND FUTURE WORK

Links has a lot in common with other functional Web programming languages and frameworks such as HOP [24], Ur/Web [10], Ocsigen [2], and Microsoft LINQ for .NET [21]. Like many of these languages, Links supports language-integrated querying, but differs in that queries are *native* (that is, use the same syntax as ordinary Links comprehension syntax, not embedded SQL or query syntax trees) and *composable* (that is, queries can be built out of reusable components). This combination of features makes Links especially useful for rapidly prototyping database-driven Web applications. Note that in contrast to Ocsigen [2], we do not use phantom types [20] to express invariants relating the SQL type system to Links; instead, these properties are directly checked in Links' type system.

Another approach to language integrated queries is Wiederman and Cook's batches [26]. A key feature of batches is that they allow some side-effecting operations to be mixed with queries, subject to certain dependency constraints. The idea is that the compiler automatically separates out the side effects from the query. The query is executed first and the results are subsequently fed into the side-effecting operations.

The Database Wiki project builds upon extensive prior work on archiving and curated databases [8, 22, 5, 4, 7]. Other projects, including SELinks [15], have used Links as a starting point for secure Web programming. SEWiki is a secure wiki system built using SELinks, which provides systematic (and provably secure) support for provenance and access control for hierarchically-structured wiki pages, but did not address querying and editing for semistructured data.

There are several other Web applications or services, such as Freebase, DabbleDB, Factual, and Google Fusion Tables [17] that aim to ease sharing, creating and visualizing data. Fusion Tables, in particular, provides many of the features of a Database Wiki: It provides limited SQL-like queries, including creating "views" that join two existing tables, as well as aggregate queries and visualizations whose results can be embedded into Web pages using JavaScript. Fusion Tables also provides some support for history and annotation. However, it lacks support for querying or editing nested (semi-)structured data and currently does not provide API access to annotations or the full history of data. These systems have a lot in common with our Database Wiki idea. However, these are closed-source systems and we aim to develop an open-source system that can in principle be used by anyone — not all scientific users will entrust a closed or commercial service with their data. There are also a number of projects to create "structured wikis" or "semantic wikis" for Semantic Web data (e.g. Semantic MediaWiki) or based on data models such as Atom feeds (e.g. AtomicWiki). However, all of these systems are aimed at specific data models and none of them support querying and editing arbitrary semistructured data through a wiki interface.

7. CONCLUSIONS

Biologists are using the Web to collaborate in creating and sharing research data, driving major advances in life sciences. However, at present they build on top of a variety of ad hoc tools, including relational databases and wikis, because no one system meets their needs, particularly concerning annotation, versioning and provenance. These applications share enough functionality to motivate developing a general-purpose system, called a Database Wiki, that combines the strengths of both relational databases and wikis.

We have constructed a pilot prototype in Links. The prototype revealed both strengths (e.g. rapid turnaround, flexible query composition and static typing) and weaknesses (e.g. performance and state handling) of Links, and helped clarify the tradeoffs between expressiveness and ease-of-use for language extensions to support database queries and updates from within wiki pages. Moreover, the DBWiki implementation can serve as a running example used to evaluate the performance and design of Links, and perhaps it can be ported to other systems and used as benchmark for other declarative approaches to Web programming.

Acknowledgments. Thanks to Peter Buneman, Henry Thompson, and Chris Yocum for discussion of this work, and to Nik Swamy for contributing the SEWiki parser. This work was supported by the University of Edinburgh via an IDEA Lab Proof of Principle Prototyping Fund award, EPSRC grant EP/F028288/1, and a Google Research Award.

8. REFERENCES

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice Hall International, 1996.

[2] Vincent Balat, Jérôme Vouillon, and Boris Yakobowski. Experience report: Ocsigen, a web programming framework. In *ICFP*, 2009.

[3] Matthias Blume, Umut A. Acar, and Wonseok Chae. Exception handlers as extensible cases. In *APLAS*, 2008.

[4] Peter Buneman. How to cite curated databases and how to make them citable. In *SSDBM*, Washington, DC, USA, 2006. IEEE.

[5] Peter Buneman, Adriane P. Chapman, and James Cheney. Provenance management in curated databases. In *SIGMOD*. ACM Press, 2006.

[6] Peter Buneman, James Cheney, Sam Lindley, and Heiko Müller. DBWiki: A structured wiki for curated databases and collaborative data management. In *SIGMOD*, 2011.

[7] Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn Vansummeren. Curated databases. In *PODS*, 2008. Invited paper.

[8] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29, 2004.

[9] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1), 1994.

[10] Adam J. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010.

[11] Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL*, 2009.

[12] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, 2007.

[13] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *APLAS*, 2008.

[14] Ezra Cooper and Philip Wadler. The RPC calculus. In *PPDP*, 2009.

[15] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*. ACM, 2009.

[16] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for

bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[17] Hector Gonzalez, Alon Y. Halevy, Christian S. Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, Warren Shen, and Jonathan Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *SIGMOD*. ACM, 2010.

[18] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: database-supported program execution. In *SIGMOD*, 2009.

[19] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5), 1997.

[20] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[21] Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[22] Heiko Müller, Peter Buneman, and Ioannis Koltsidas. XArch: archiving scientific and reference data. In *SIGMOD*, 2008.

[23] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[24] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Dynamic Languages Symposium*, New York, NY, USA, 2006. ACM.

[25] Alexander Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011.

[26] Ben Wiedermann and William R. Cook. Remote batch invocation for SQL databases. In *DBPL*, 2011.

[27] Limsoon Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1), 2000.