Dependent Types for Safe and Secure Web Programming

Simon Fowler Edwin Brady

School of Computer Science, University of St Andrews, St Andrews, Scotland
Email: {sf37, ecb10}@st-andrews.ac.uk

Abstract

Dependently-typed languages allow precise types to be used during development, facilitating reasoning about programs. However, stronger types bring a disadvantage that it becomes increasingly difficult to write programs that are accepted by a type checker and additional proofs may have to be specified by a programmer.

Embedded domain-specific languages (EDSLs) can help address this problem by introducing a layer of abstraction over more precise underlying types, allowing domain-specific code to be written in a high-level language which uses dependent types to enforce invariants without imposing additional proof obligations on an application programmer.

In this paper, we apply this technique to web programming. Using the dependently typed programming language IDRIS, we introduce an EDSL to facilitate the creation and handling of *statically checked* web forms, reducing the scope for programmer error and attacks such as SQL injection. We also show how to enforce resource usage protocols associated with common web operations such as CGI, database access and session handling.

1. Introduction

Web applications, whilst ubiquitous, are also prone to incorrect construction and security exploits such as SQL injection [12] or cross-site scripting [11]. Security breaches using such exploits are far-reaching, and high profile cases involve large corporations such as Sony, who suffered a well-publicised and extremely costly SQL Injection breach in 2011 [7], and *Yahoo!*, who suffered a breach in 2012 [8].

Many web applications are written in dynamically-checked scripting languages such as PHP, Ruby or Python, which facilitate rapid development [20]. However, such languages do not provide the same static guarantees about runtime behaviour afforded by programs with more expressive, static type systems, instead relying on extensive unit testing to ensure correctness and security.

Let us consider a simple database access routine, written in PHP, where we wish to obtain the name and address of every employee working in a given department, `$dept`. We firstly construct an object representing a database connection, where the arguments are the database host, user, password and name respectively:

```
$conn = new mysqli("localhost", "username",
                   "password", "db");
```

We should then check to see if the connection was successful, and exit if not:

```
if (mysqli_connect_errno()) { exit(); }
```

We then create a prepared statement detailing our query, and bind the `'dept'` value:

```
$stmt = $conn->prepare("SELECT `name`, `address`
   FROM `staff` WHERE `dept` = ?);
$stmt->bind_param('s', $dept);
```

After the parameters have been bound, we execute the statement, assign variables into which results will be stored, and fetch each row in turn. Failure to execute a statement before attempting to fetch rows would cause an error, as would attempting to execute a statement without binding variables to it.

```
$stmt->execute();
$stmt->bind_result($name, $address);
while ($stmt->fetch()) {
  printf("Name: %s, Age: %s", $name, $age);
}
```

Finally, once the statement and connection are no longer needed, they should be closed in order to discard the associated resources:

```
$stmt->close();
$conn->close();
```

Even in this small example, there exists a precise resource usage protocol which must be followed for successful and robust operation. Firstly, a connection to the database must be opened. The object-oriented style used in the example encapsulates this to an extent, as the object must be created in order for operations to be performed, however it is less obvious in a procedural version of the code. Secondly, a prepared statement is created, using the raw SQL and placeholders to which variables are later bound. The statement is then executed, and each row is retrieved from the database. Finally, the resources are freed.

Problems may arise if the protocol is not followed correctly. A developer may, for example, accidentally close a statement whilst still retrieving rows, which would cause a runtime error. Similarly, a programmer may omit closing the statement or connection, which can lead to problems such as resource leaks in longer-running server applications. However, in conventional programming languages, there is no way to check automatically, at compile-time, that a protocol is followed.

In contrast, the use of *dependent types* makes it possible to specify a program's behaviour precisely, and to check that a specification is followed. Unfortunately, automatic verification by a compiler can be difficult or often impossible, requiring additional proofs to be given by the programmer.

This complexity can be addressed through the use of *embedded domain-specific languages* (EDSLs) to abstract away the complexity of the underlying type system. Embedding a domain-specific language in a dependently typed host language allows domain experts to write domain-specific code, with the EDSL itself used to provide the proof that the code is correct.

IDRIS [2] is a language with full dependent types, and extensive support for EDSLs through overloading and syntax macros. Through the use of IDRIS, and a framework for describing resource protocols using *algebraic effects* [3], we present a dependently-typed web framework, which allows the construction of programs with additional guarantees about correctness and security, whilst minimising the increase in development complexity.

1.1 Contributions

The primary contribution of this paper is the application of dependent types to provide strong static guarantees about the correctness and security of web applications, whilst minimising additional development complexity. In particular, we present:

- Representations of CGI, Databases and sessions as *resource-dependent algebraic effects*, allowing programs to be accepted only when they follow clearly defined resource usage protocols. (Section 3)
- Type-safe form handling, preserving type information and managing user input, therefore increasing applications' resilience to attacks such as SQL injection and cross-site scripting. (Section 4)
- An extended example: a message board application, demonstrating the usage of the framework in practice. (Section 2)

We achieve these without extending the host language. Every resource protocol we implement is pure IDRIS, using a library for resource-dependent algebraic effects [3] and IDRIS' features for supporting domain-specific language implementation such as syntax macros and overloading. In particular, this means the same techniques can be applied to other resources, and most importantly, combinations of resources and DSLs implemented in this way are *composable*.

The code used to implement the framework and all associated examples used in this paper is available online at `http://www.github.com/idris-hackers/IdrisWeb`.

2. An overview of the `Effects` framework

`Effects` [3] is an IDRIS library which handles side-effects such as state, exceptions, and I/O as *algebraic effects* [16]. In particular, it supports parameterising effects by an input and output *state*, which permits effectful programs to track the progress of a resource usage protocol. Effectful programs are written in a monadic style, with `do`-notation, with their type stating which specific effects are available. Effectful programs are described using the following data type, in the simplest case:

```
Eff : (m : Type -> Type) ->
      (es : List EFFECT) -> (a : Type) -> Type
```

`Eff` is parameterised over a *computation context*, `m`, which describes the context in which the effectful program will be run, a list of side effects `es` that the program is permitted to use, and the programs return type `a`. The name `m` for the computation context is suggestive of a monad, but there is no requirement for it to be so.

For example, the following type carries an integer state, throws an exception of type `String` if the state reaches 100, and runs in a `Maybe` context:

```
addState : Eff Maybe [STATE Int, EXCEPTION String] ()
addState = do val <- get
              when (val == 100) (raise "State too big")
              put (val + 1)
```

2.1 Implementing Effects

Effects such as state and exception are described as algebraic data types, and run by giving *handlers* for specific computation contexts. Effects have a corresponding *resource* (in the case of state, the resource is simply the current state). Executing an effectful operation may change the resource and return a value:

```
Effect : Type
Effect = (in_res : Type) -> (out_res : Type) ->
         (val : Type) -> Type
```

For example, the state effect is described as follows:

```
data State : Effect where
     Get :      State a a a
     Put : b -> State a b ()
```

That is, `Get` returns a value of type `a` without updating the resource type. `Put` returns nothing, but has the effect of updating the resource. To make an effect usable, we implement a handler for a computation context by making an instance of the following class:

```
class Handler (e : Effect) (m : Type -> Type) where
     handle : res -> (eff : e res res' t) ->
              (k: res' -> t -> m a) -> m a
```

The `handle` function takes the input resource, an effect which may update that resource and execute a side-effect, and a continuation `k` which takes the updated resource and the return value of the effect. We use a continuation here primarily because there is no restriction on the number of times a handler may invoke the continuation (raising an exception, for example, will not invoke it). Reading and updating states is handled for all computation contexts `m`:

```
instance Handler State m where
     handle st Get     k = k st st
     handle st (Put n) k = k n ()
```

Finally, we promote `State` into a concrete effect `STATE`, and the `Get` and `Put` operations into functions in `Eff`, as follows:

```
STATE : Type -> EFFECT
STATE t = MkEff t State

get : Eff m [STATE x] x
get = Get

put : x -> Eff m [STATE x] ()
put val = Put val
```

A concrete effect is simply an algebraic effect type paired with its current resource type. This, and other technical details, are explained in full elsewhere [3]. For the purposes of this paper, it suffices to know how to describe and handle new algebraic effects.

2.2 Resource Protocols as Effects

More generally, a program might modify the set of effects available. This might be desirable for several reasons, such as adding a new effect, or to update an index of a dependently typed state. In this case, we describe programs using the `EffM` data type:

```
EffM : (m : Type -> Type) ->
       (es : List EFFECT) -> (es' : List EFFECT) ->
       (a : Type) -> Type
```

`EffM` is parameterised over the context and type as before, but separates input effects (`es`) from output effects (`es'`). In fact, `Eff`

```
{-                                        { Input resource type }   { Output resource type } { Value } -}

data FileIO : Effect where
     Open      : String -> (m : Mode) -> FileIO     ()                  (Either () (OpenFile m)) ()
     Close     :                         FileIO     (OpenFile m)        ()                       ()

     ReadLine  :                         FileIO     (OpenFile Read)     (OpenFile Read)          String
     WriteLine : String ->               FileIO     (OpenFile Write)    (OpenFile Write)         ()
     EOF       :                         FileIO     (OpenFile Read)     (OpenFile Read)          Bool
```

Figure 1. File Protocol Effect

is defined in terms of `EffM`, with equal input/output effects. We can use this to describe how effects follow resource protocols. A simple example is a file access protocol, where a file must be opened before it is read or written, and a file must be closed on exit.

Figure 1 shows how the protocol is encoded as an effect. Note that the types of the input and output resources describes how resource state changes in each operation: opening a file may fail, so changes an empty resource to a resource containing either a unit or an open file; reading a line is only possible if the resoource is a file open for reading, etc. The handler for this effect for an `IO` computation context will execute the required primitive I/O actions.

The following program type checks, and therefore implicitly carries a proof that the file resource protocol is followed correctly:

```
testFile : Eff IO [FILE_IO (), STDIO] ()
testFile = do open "testFile" Read
              if_valid then do str <- readLine
                               close
                               putStrLn str
                       else putStrLn "Error"
```

The type of `testFile` states that File I/O and console I/O are available effects, and in particular that the resource associated with the File I/O will be in the same state on entry and exit. We use `if_valid` to handle possible failure—this is a function provided by the `Effects` library which checks whether a resource of type `Either a b` indicates failure (`Left a`) or success (`Right b`) and branches and updates the resource accordingly. Therefore, attempting to write to the file, failing to check for success, or failing to open or close the file, would cause a *compile-time* error.

We will use this technique extensively throughout this paper: describe a resource usage protocol in terms of state transitions; implement an effect which captures that protocol; implement programs which, by using this effect, implicitly carry a proof that the resource protocol has been correctly followed.

3. Modelling resource usage protocols

In this section, we show how three effects (CGI, database access and a simple session handler) may be implemented, and describe the benefits of developing programs using this technique over simply handling them in `IO` or as monad transformers.

3.1 CGI

CGI is used to invoke an application on a web server, making use of environment variables to convey information gained from an HTTP request and using standard output to communicate with the remote client. Importantly, HTTP headers must be correctly written to the browser prior to any other output; failure to do so will result in an internal server error being shown.

By modelling CGI as a resource-dependent algebraic effect, we may enforce a resource usage protocol which prevents arbitrary IO from being performed and therefore ensures that the headers are written correctly. We define an effect, `Cgi`, and an associated resource, `InitialisedCGI`, parameterised over the current state,

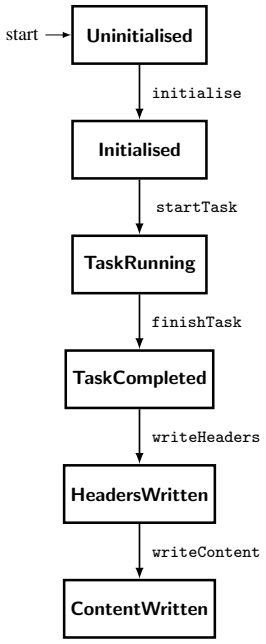

Figure 2. CGI States

`CGIStep`, and containing a `CGIInfo` record which contains information from the request. We represent an uninitialised CGI process as the unit type, `()`.

```
data CGIStep = Initialised   | TaskRunning
             | TaskCompleted | HeadersWritten
             | ContentWritten

data InitialisedCGI : CGIStep -> Type where
     ICgi : CGIInfo -> InitialisedCGI s
```

Figure 2 shows the states through which the CGI program progresses, and Figure 3 shows how this is represented as a resource-dependent algebraic effect. Each operation performed in an effectful program requires the resource to be of a certain type, and the completion of the operation may alter the type or value of the resource. The `Cgi` effect declaration shows these resource updates in the types of each operation, effectively specifying a state machine.

Upon creation, the CGI application is uninitialised, meaning that environment variables have not been queried to populate the CGI state. The only operation that can be performed in this state is initialisation: by calling `initialise`, a `CGIInfo` record is populated, and the state transitions to `Initialised`. The `Init` operation is defined as part of the `Cgi` effect, and involves transitioning from the uninitialised state to the initialised state.

Additional operations, including those to query `POST` and `GET` variables, are omitted in the interest of brevity.

```
{-                    { Input resource type }         { Output resource type }        { Value } -}

data Cgi : Effect where
    Init         : Cgi ()                                (InitialisedCGI Initialised)    ()
    StartRun     : Cgi (InitialisedCGI Initialised)      (InitialisedCGI TaskRunning)    ()
    FinishRun    : Cgi (InitialisedCGI TaskRunning)      (InitialisedCGI TaskCompleted)  ()
    WriteHeaders : Cgi (InitialisedCGI TaskCompleted)    (InitialisedCGI HeadersWritten) ()
    WriteContent : Cgi (InitialisedCGI HeadersWritten)   (InitialisedCGI ContentWritten) ()
    OutputData   : String ->
                   Cgi (InitialisedCGI TaskRunning)      (InitialisedCGI TaskRunning)    ()
    RunAction    : Env IO (CGI (InitialisedCGI TaskRunning) :: effs) -> CGIProg effs a ->
                   Cgi (InitialisedCGI TaskRunning)      (InitialisedCGI TaskRunning)    a
```

Figure 3. CGI Effect

User code executes in the `TaskRunning` state. Several operations, such as querying the `POST` and `GET` variables, are available in this state, alongside functions to output data to the web page and append data to the response headers. It is important to note that at this stage nothing is written to the page, with the `output` and `addHeader` functions instead modifying the `CGIInfo` record. This data may then be printed at the end of the program's execution, in accordance with the resource usage protocol.

After the user code has finished execution, control returns to the library code. At this point, the state transitions to TaskCompleted, and the headers are written. Finally, the headers and content are written which completes the process. Since we parameterise the resource over a state, we may ensure that certain operations only happen in a particular prescribed order.

In IDRIS, types are first-class, meaning that they may be treated like other terms in computations. We may therefore define the following type synonym, used within the CGI section of the framework to denote an effectful CGI program:

```
CGIProg : List EFFECT -> Type -> Type
CGIProg effs a =
  Eff IO (CGI (InitialisedCGI TaskRunning) :: effs) a
```

This is then passed, along with initial values for other effects that the user may wish to use, to the runAction function, which invokes the RunAction operation and executes the user-specified action. A simple "Hello, world!" program would be defined as follows:

```
module Main
import Cgi

sayHello : CGIProg [] ()
sayHello = output "Hello, world!"

main : IO ()
main = runCGI [initCGIState] sayHello
```

3.2 Database access with SQLite

SQLite[1] is a lightweight SQL database engine often used as simple, structured storage for larger applications. We make use of SQLite to demonstrate a resource usage protocol for database access due to its simplicity, although we envisage that these concepts would be applicable to more complex database management systems.

The creation, preparation and execution of SQL queries has a specific usage protocol, with several possible points of failure. Failure is handled in traditional web applications by the generation of exceptions, which may be handled in the program. Handling such exceptions is often optional, however, and in some cases unhandled errors may cause a deployed web application to display an error to the user. Such errors can be used to determine the structure of an insecure SQL query, and are often used by attackers to determine attack vectors for SQL injection attacks.

[1] `http://www.sqlite.org`

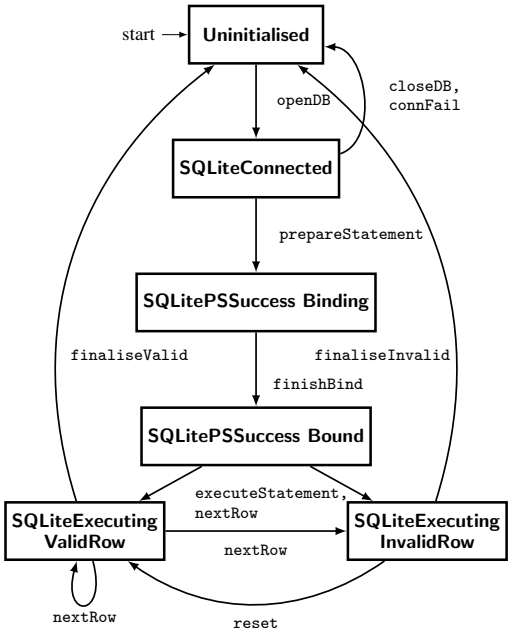

Figure 5. Database Resource Usage Protocol

Figure 5 shows a resource usage protocol for database access, which we have implemented for the SQLite library. Although some additional states are used to capture failing computations, these are omitted from the diagram. The effect implementation is given in Figure 4. There are three main phases involved in the usage of the SQLite protocol: connection to the database, preparation of a query, and execution of the query. We define several resources to encapsulate the state at any given point during the protocol.

We first define the `SQLiteConnected` resource, which signifies that a successful connection has been made to the database. This resource contains a pointer to the database structure which is used in further computations.

```
data SQLiteConnected : Type where
  SQLConnection : ConnectionPtr -> SQLiteConnected
```

We secondly define resource types to capture success and failure states of binding a prepared statement: `SQLitePSSuccess`, `SQLitePSFail`, and `SQLiteFinishBindFail`. The types are declared as follows (we leave the definitions abstract):

```
{-                          { Input resource type }    { Output resource type }             { Value }           -}
data Sqlite : Effect where
  OpenDB            : DBName ->
                      Sqlite ()                        (Either () SQLiteConnected)          (Either SQLiteCode ())
  CloseDB           : Sqlite (SQLiteConnected)         ()                                   ()
  PrepareStatement  : QueryString ->
                      Sqlite (SQLiteConnected)         (Either (SQLitePSFail)
                                                         (SQLitePSSuccess Binding))         (Either SQLiteCode ())
  BindInt           : ArgPos -> Int ->
                      Sqlite (SQLitePSSuccess Binding) (SQLitePSSuccess Binding)            ()
  FinishBind        : Sqlite (SQLitePSSuccess Binding) (Either SQLiteFinishBindFail
                                                         (SQLitePSSuccess Bound))           (Maybe BindError)
  ExecuteStatement  : Sqlite (SQLitePSSuccess Bound)   (Either (SQLiteExecuting InvalidRow)
                                                         (SQLiteExecuting ValidRow)         StepResult
  RowStep           : Sqlite (SQLiteExecuting ValidRow) (Either (SQLiteExecuting InvalidRow)
                                                         (SQLiteExecuting ValidRow))        StepResult
  GetColumnText     : Column ->
                      Sqlite (SQLiteExecuting ValidRow) (SQLiteExecuting ValidRow)          String

  CleanupPSFail     : Sqlite (SQLitePSFail)            ()                                   ()
  CleanupBindFail   : Sqlite (SQLiteFinishBindFail)    ()                                   ()
```

Figure 4. Database Effect

```
data BindStep = Binding | Bound
data SQLitePSSuccess : BindStep -> Type where
data SQLitePSFail : Type where
data SQLiteFinishBindFail : Type where
```

The `SQLitePSSuccess` resource indicates that a prepared statement has been correctly created by the underlying library, given a query string. The resource is parameterised by the `BindStep` type, which indicates whether the binding process has been completed. Should the creation of a prepared statement fail, the resource will be set to `SQLitePSFail`.

SQLite operates by loading database rows into a buffer, the contents of which may then be accessed through several column access functions. After all rows returned by a query have been processed, no further calls to fetch more rows may be made. Additionally, no calls to column access functions may be made whilst there is no row being currently processed.

Through the use of dependent types, we may encode these invariants statically, using the `SQLiteExecuting` resource. This is parameterised by the `ExecutionResult` data type, which indicates whether there is currently a valid row in the buffer.

```
data ExecutionResult = ValidRow
                     | InvalidRow

data SQLiteExecuting : ExecutionResult -> Type where
  SQLiteE : ConnectionPtr ->
            StmtPtr -> SQLiteExecuting a
```

Column access functions may only be called when there is a valid row in the buffer, as signified by the input resources:

```
GetColumnText : Column ->
                Sqlite (SQLiteExecuting ValidRow)
                       (SQLiteExecuting ValidRow)
                       String
```

In order to provide the necessary static guarantee that there is a valid row to process in the buffer, we make use of the `if_valid` construct. The `nextRow` function will either fetch a row into the buffer, or indicate that a row could not be loaded, as shown by its type:

```
nextRow : EffM IO [SQLITE (SQLiteExecuting ValidRow)]
          [SQLITE (Either (SQLiteExecuting InvalidRow)
                          (SQLiteExecuting ValidRow))] StepResult
```

The `if_valid` construct provides failure checking functionality, allowing different operations to be performed depending on whether or not a row was successfully fetched for processing.

By incorporating pointers to open connections and prepared statements into the resource associated with the effect, we introduce a further layer of abstraction, which hides implementation details from the developer and encourages less error-prone code.

3.2.1 Example

To demonstrate the library, we return to the previous example of selecting the names and addresses of all staff working in a given department. We define a function `textSel` of type:

```
String -> Eff IO [SQLITE ()]
          (Either QueryError (List (String, String)))
```

The program will be run in `IO`, and starts and finishes with no active resources. It returns either a list of `(String, String)` pairs, representing names and addresses in the database, or an error.

The program initially attempts to open a connection to the `people.db` database. At this point, since the `OpenDB` operation has been invoked, the program transitions to the `ConnectionOpened` state. The `openDB` function returns either an error code, if the connection fails, or a unit type should the connection succeed, as shown in Figure 5.

A call to `prepareStatement` attempts to create a prepared statement, and a subsequent call to `beginExecution` allows data to be retrieved from the database. `CollectResults` operates on the row currently held in the buffer. Firstly, the function checks that there is a valid row in the buffer, and if so uses the `getColumnText` and `getColumnInt` functions to retrieve the data from the database. This function is then called recursively until there are no more rows to process.

```
collectResults :
  EffM IO [SQLITE (Either (SQLiteExecuting InvalidRow)
                          (SQLiteExecuting ValidRow))]
          [SQLITE (SQLiteExecuting InvalidRow)]
          (List (String, Int))
```

```
testSelect : String -> Eff IO [SQLITE ()]
             (Either QueryError (List (String, String)))
testSelect dept = do
  db_res <- openDB "people.db"
  if_valid then do
    let sql = "SELECT name, address FROM ‘staff‘
                   WHERE dept = ?;"
    sql_prep_res <- prepareStatement sql
    if_valid then do
      bindText 1 dept
      bind_res <- finishBind
      if_valid then do
        executeStatement
        results <- collectResults
        finaliseInvalid
        closeDB
        Effects.pure $ Right results
      else do
        cleanupBindFail
        Effects.pure $ Left (getBindError bind_res)
    else do
      cleanupPSFail
      Left $ getQueryError sql_prep_res
  else do
    Effects.pure $ Left (getQueryError db_res)
```

Figure 6. Example SQL program

```
collectResults =
  if_valid then do name <- getColumnText 1
                   age <- getColumnInt 2
                   step_res <- nextRow
                   xs <- collectResults
                   return $ (name, age) :: xs
  else return []
```

Using this, we can build a function to execute a full query:

```
executeSelect :
  String -> String -> List (Int, DBVal) ->
  (Eff IO [SQLITE (SQLiteExecuting ValidRow)]
              (List DBVal)) ->
  Eff IO [SQLITE ()] (Either QueryError ResultSet)
```

This returns either an error, or a set of results, defined as follows:

```
ResultSet : Type
ResultSet = List (List DBVal)

data DBVal = DBInt Int     | DBText String
           | DBFloat Float | DBNull
```

3.3 A Simple Session Handler

Larger web applications require persistent state across separate requests. This can be achieved using a *session*, in which a cookie is set on the remote host containing a unique session ID, which is used to retrieve data. In this section, we describe a simple session handler, and the resource protocol involved.

The `Effects` library allows for composition of individual, fine-grained effects. By combining the `CGI` and `SQLite` components, we can construct a simple session handler to provide a notion of state across separate web requests. We implement this with a SQLite database containing tables for storing session keys and expiry dates, along with the data associated with the session.

Figure 8 shows the resource usage protocol associated with the session handler, and Figure 7 the corresponding algebraic effect. In this application, there are two states: In `SessionClosed`, the user may load or create a session. In `SessionOpen`, the user may update the representation of the session in memory, serialise the session and write it to the database, or delete the session and invalidate

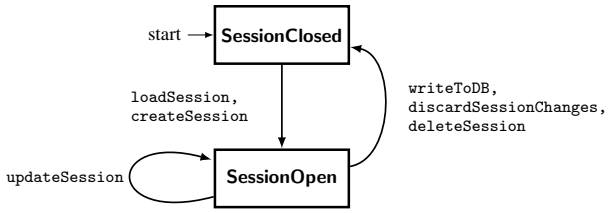

Figure 8. Session Handler Resource Usage Protocol

the user's session key. These two states ensure that changes are explicitly either written or discarded, eliminating the possibility of a developer updating the session but neglecting to commit it to persistent storage. This, of course, is under the assumption that the process exits cleanly: we attempt to facilitate this by writing total functions where possible.

Much like the SQLite effect, we encapsulate failure by reflecting it in the resource associated with the effect.

```
data SessionStep = SessionClosed | SessionOpen
data SessionRes : SessionStep -> Type where
  InvalidSession : SessionRes s
  ValidSession   : SessionID ->
                   SessionData ->
                   SessionRes s
```

4. Type-aware form handling

Programming web applications often involves processing user data, which may then be used in further effectful computations. Data submitted using a form is transmitted as a string as part of an HTTP request, which traditionally involves losing associated type information.

This can in turn lead to risks; developers may assume that data is of a certain type, and therefore discount the possibility that it may have been modified by an attacker. One example would be the traversal of paginated data, in which a form is used to make a request to retrieve the next page of data. This may involve sending an integer detailing the current page, which could be used in a query such as:

```
’SELECT ‘name‘, ‘address‘ FROM ‘staff‘ LIMIT ’ +
       page + ’, 5’;
```

The `page` variable is assumed to be an integer, but may instead be modified by an attacker to include a malicious string which would alter the semantics of the query, allowing an attacker to execute a blind SQL injection attack.

In this section, we present a DSL for the creation of web forms which preserve type information, implemented as a resource-dependent algebraic effect. Once the form has been submitted, retrieved information is passed directly to a developer-specified function for handling, without the need to manually check and deserialise data.

We begin with a simple example of a form which requests a user's name, and echoes it back. Firstly, we define a form handler which echoes back a string provided by the form handler. It has one argument of type `Maybe String`, which accounts for the possibility that the user may have provided invalid data:

```
echo : Maybe String ->
     FormHandler [CGI (InitialisedCGI TaskRunning)]
echo (Just name) = output ("Hello, " ++ name ++ "!")
echo _ = output "Error!"
```

We then specify this in a list of handlers, detailing the arguments, available effects, handler function and unique identifier:

```
{-                        { Input resource type }           { Output resource type }   { Value }        -}

data Session : Effect where
  LoadSession           : SessionID ->
                          Session (SessionRes SessionClosed) (SessionRes SessionOpen)   (Maybe SessionData)
  UpdateSession         : SessionData ->
                          Session (SessionRes SessionOpen)   (SessionRes SessionOpen)   ()
  CreateSession         : SessionData ->
                          Session (SessionRes SessionClosed) (SessionRes SessionOpen)   (Maybe SessionID)
  DeleteSession         : Session (SessionRes SessionOpen)   (SessionRes SessionClosed) Bool
  WriteToDB             : Session (SessionRes SessionOpen)   (SessionRes SessionClosed) Bool
  DiscardSessionChanges : Session (SessionRes SessionOpen)   (SessionRes SessionClosed) ()
  GetSessionID          : Session (SessionRes SessionOpen)   (SessionRes SessionOpen)   (Maybe SessionID)
  GetSessionData        : Session (SessionRes SessionOpen)   (SessionRes SessionOpen)   (Maybe SessionData)
```

Figure 7. Session Effect

```
handlers : HandlerList
handlers = [handler args=[FormString],
                    effects=[CgiEffect],
                    fn=echo,
                    name="echo"]
```

We also define a form to take in a name from the user, and specify that it should use the `echo` handler.

```
showHelloForm : UserForm
showHelloForm = do
  addTextBox "Name" FormString Nothing
  useEffects [CgiEffect]
  addSubmit echo handlers
```

Finally, we specify that if data has been submitted for processing, then it should be passed to the form handler. If not, then the form should be shown.

```
cgiHello : CGIProg [] ()
cgiHello = do
  handler_set <- isHandlerSet
  if handler_set then do
    handleForm handlers
    return ()
  else do
    addForm "nameform" "helloform" showHelloForm
    return ()

main : IO ()
main = runCGI [initCGIState] cgiHello
```

When this CGI application is invoked, it will begin by outputting a form to the page, requesting a name from the user. Upon submission of the form, the form handler will be invoked, and the name will be used in the output.

In Sections 4.1 and 4.2, we examine implementation of the form-handling system: namely, the effect which allows the creation of forms, and the handling code which deserialises the data and passes it to the user-specified handler function.

4.1 Form Construction

Each form element is specified to hold a particular type of data, which, assuming that the correct type of data is specified by the user, is passed directly to the handler function. In order to encapsulate this, we firstly define the allowed data types as part of an algebraic data type, `FormTy`.

```
data FormTy = FormString | FormInt
            | FormBool   | FormFloat
            | FormList FormTy
```

Since types in IDRIS are first-class, we may use this to convert between abstract and concrete representations of allowed form types:

```
interpFormTy : FormTy -> Type
interpFormTy FormString = String
interpFormTy FormInt = Int
interpFormTy FormBool = Bool
interpFormTy FormFloat = Float
interpFormTy (FormList a) = List (interpFormTy a)
```

Again, we use `Effects` to build a form. By recording the type of each form element as it is added in the type of the form, we may statically ensure that the user-supplied handler function is of the correct type to handle the data supplied by the form: using an incompatible handler will result in a compile-time type error. The `Form` effect and associated resource `FormRes` is given in Figure 9. The `using` notation here indicates that within the block, where `G` and `E` occur they are implicit arguments with the given type.

The general process of form construction is illustrated by the `AddTextBox` and `Submit` operations of the `Form` effect:

```
data Form : Effect where
  AddTextBox : (label : String) -> (fty : FormTy) ->
               Maybe (interpFormTy fty) ->
               Form (FormRes G E)
                    (FormRes (fty :: G) E) ()
    ...
  Submit : (mkHandlerFn ((reverse G), E)) -> String ->
           Form (FormRes G E) (FormRes [] []) String
```

These use the resource associated with the effect, `FormRes`, to construct the form. Adding a field such as a text box adds a new type, `fty` to the list of field types, carried in the resource. When the form is complete, the `Submit` operation adds a submit button and returns the HTML text for the form, flushing the list of field types, and using it to construct the type for an appropriate handler function. To specify a form instance, we define a function of type `UserForm`:

```
UserForm : Type
UserForm = EffM m [FORM (FormRes [])
                        (FormRes [])] String
```

The input and output resource contains an empty list of types, which means that any form which includes fields must also include a submit button. Adding fields adds to the list of types, and only adding a submit button empties that list. Note that there is no need to restrict this effect to running in the `IO` monad since creating a form merely returns HTML text, with no side-effects by default.

Handlers may only be associated with a form if they have argument types corresponding to the types associated with the form elements. Additionally, we wish to name the function in order for it to be serialised, whilst requiring a proof that the specified name is associated with the function. If this were not required, it would be

```
using (G : List FormTy, E : List WebEffect)
  data FormRes : List FormTy -> List WebEffect -> Type where
    FR : Nat -> List FormTy -> List WebEffect -> String -> FormRes G E

  data Form : Effect where
    AddTextBox      : (label : String) -> (fty : FormTy) -> (Maybe (interpFormTy fty)) ->
                      Form (FormRes G E) (FormRes (fty :: G) E) ()
    AddSelectionBox : (label : String) -> (fty : FormTy) -> (vals : Vect m (interpFormTy fty)) ->
                      (names : Vect m String) ->
                      Form (FormRes G E)  (FormRes (fty :: G) E) ()
    AddRadioGroup   : (label : String) -> (fty : FormTy) -> (vals : Vect m (interpFormTy fty)) ->
                      (names : Vect m String) -> (default : Int) ->
                      Form (FormRes G E)  (FormRes (fty :: G) E) ()
    AddCheckBoxes   : (label : String) -> (fty : FormTy) -> (vals : Vect m (interpFormTy fty)) ->
                      (names : Vect m String) -> (checked_boxes : Vect m Bool) ->
                      Form (FormRes G E)  (FormRes ((FormList fty) :: G) E) ()
    Submit          : (mkHandlerFn ((reverse G), E)) -> String ->
                      Form (FormRes G E)  (FormRes [] [])       String
```

Figure 9. Form Effect

possible to use a function satisfying the type requirement, without guaranteeing that the serialised data corresponded to that function.

Before associating a handler function with the form, we must specify the effects available to the handler. This is done with `useEffects`, which updates the list of effects in the type of the form resource. By doing this, we may subsequently use the effects in calculations at the type level, in particular when calculating the type of the handler function for the form.

```
useEffects : (effs : List WebEffect) ->
             EffM m [FORM (FormRes G E)]
                    [FORM (FormRes G effs)] ()
useEffects effs = (UseEffects effs)
```

A `WebEffect` is an effect which is usable in a web application, and can be converted to an `EFFECT` using:

```
webEffect : WebEffect -> EFFECT
```

Whilst it is not possible to serialise arbitrary effects due to the associated difficulties with serialising initial resource environments, we allow for three effects to be serialised: `CGI`, `SQLITE` and `SESSION`. This is, however, not an inherent limitation as the `Effects` library permits introduction of additional effects within an effectful computation. We may specify a handler function of type `FormHandler`:

```
FormHandler : List EFFECT -> Type
FormHandler effs = Eff IO effs ()
```

In order to associate a handler with a form, we may call the `addSubmit` function:

```
addSubmit : (f :  mkHandlerFn ((reverse G), E)) ->
            (fns : HandlerList) ->
            {default tactics
              { applyTactic findFn 100; solve; }
              prf : FnElem f fns} ->
            EffM m [FORM (FormRes G E)]
                   [FORM (FormRes [] [])]
                   String
addSubmit f handlers {prf} = (Submit f name)
    where name : String
          name = getString' f handlers prf
```

This function takes a handler function and a list of available handlers, along with an automatically constructed proof (using the a `default` argument) that the handler is available. Let us look at each aspect of this function in turn. Firstly, the `mkHandlerFn` function calculates the required type of the handler function from the list of types associated with the form elements, and the effects we specified with `useEffects`. Note that since we prepend types to the list of `FormTy`s as opposed to appending them, we must reverse the list.

```
MkHandlerFnTy : Type
MkHandlerFnTy = (List FormTy, List WebEffect)

mkHandlerFn' : List FormTy -> List WebEffect -> Type
mkHandlerFn' [] effs = FormHandler (map webEffect effs)
mkHandlerFn' (x :: xs) effs = Maybe (interpFormTy x) ->
                              mkHandlerFn' xs effs
mkHandlerFn : MkHandlerFnTy -> Type
mkHandlerFn (tys, effs) = mkHandlerFn' tys effs
```

The `mkHandlerFn` function takes a tuple describing the arguments and web effects available to the handler function. When constructing the function type, we wrap all arguments in a `Maybe`, in order to handle failure should the supplied data fail to parse as the required type. To store a reference to a handler function, we use the `HandlerFn` type:

```
HandlerFn : Type
HandlerFn = (ft ** (mkHandlerFn ft, String))
```

The `**` notation denotes a dependent pair, in which the type of the second element of the pair is parameterised over the value of the first element. It is an existential binding: the notation `(x ** P x)` can be read as "there exists an `x` such that `P x` holds". Therefore a `HandlerFn` states that there exists a function type `ft` such that we have a handler for it, and a unique string identifier which is used to serialise a reference to the handler function.

In order to abstract away from this implementation detail, we make use of IDRIS syntax rewriting rules. This allows us to define the following:

```
syntax
  "handler args=" [args] ", effects=" [effs] ", fn=" [fn]
  ", name=" [name] = ((args, effs) ** (fn, name))
```

We may then define handlers in a more intuitive fashion, without being concerned with the implementation details. This allows us to write a handler with one String argument, making use of the CGI effect, associated with the `echo` handler function as follows:

```
handler args=[FormString],
        effects=[CgiEffect],
        fn=echo,
        name="echo"
```

We then store each `HandlerFn` in a `HandlerList`.

```
HandlerList : Type
HandlerList = List HandlerFn
```

To enforce the requirement that a supplied handler function must be in the list of available handlers, and therefore allow us to retrieve the name with which to serialise the handler, we require a *list membership proof*, `FnElem f fns`, which statically guarantees that a given item resides in a list.

```
using (xs : HanderList , f : mkHandler (reverse G, E))
  data FnElem : mkHandlerFn ((reverse G), E) ->
                HandlerList -> Type where
       FnHere  : FnElem f (((reverse G, E) **
                           (f, fStr)) :: xs)
       FnThere : FnElem f xs -> FnElem f (x :: xs)
```

`FnElem` is parameterised over `G` and `E`, the types of the form elements and the effects used by the handler function. `FnHere` is a proof that the element is at the head of the current point of the list, whereas `FnThere` is a proof that the element is in the tail of the list. We then use an automatic proof search to generate the proof at compile time, should one exist. The proof may then be used in subsequent computations: we use it to retrieve the unique identifier for the function. If the automated proof search fails, compilation will fail.

Finally, we serialise the argument types, supported effects, and return type of the handler, to allow the form data to be deserialised and ensure that the correct handler is executed on the server.

Although sending details of the handler function to the client may appear to be a security risk, we envisage that the use of symmetric encryption or a cryptographic nonce would mitigate this. Ultimately, we hope to implement a web server with persistent state, which would eliminate the need for serialisation altogether.

Running form construction is achieved as an operation of the `CGI` effect, `AddForm`, which then outputs the generated HTML to the page. The generated metadata describing the handler function is serialised as a hidden HTML field.

4.2 Form Handling

Once the form has been submitted, a web application may handle the submitted data by invoking `HandleForm`. This will check for the existence of the hidden `handler` field, which contains the previously serialised metadata about the form handler, before deserialising the data into a `MkHandlerFnTy`.

With this data, we then look up the function in the list of registered handlers by using the unique handler identifier. In order to apply the handler function to the data submitted in the form, we must first prove to the type checker that the deserialised `MkHandlerFnTy` is the same as the one retrieved from the list of registered handlers. We do this by making use of the `decEq` function, which determines whether two types are equal, returning a proof of equality if so, and a proof of inequality if not.

```
decEq : DecEq t => (x : t) -> (y : t) -> Dec (x = y)
```

We then use the `with` construct, inspired by *views* in Epigram [9], to rewrite the arguments on the left hand side. This allows us to construct a function which, given the stored handler, the data required to construct the function type and the `MkHandlerFnTy` deserialised from the form, determines whether the two `MkHandlerFnTy`s are decidably equal. If so, we rewrite this on the left hand side since the equality proof demonstrates that the recorded function may also be used to handle the form data. If not, the computation fails.

```
checkFunctions : (reg_fn_ty : MkHandlerFnTy) ->
                 (frm_fn_ty : MkHandlerFnTy) ->
                 mkHandlerFn reg_fn_ty ->
                 Maybe (mkHandlerFn frm_fn_ty)
checkFunctions reg_ty frm_ty reg_fn with
                              (decEq reg_ty frm_ty)
  checkFunctions frm_ty frm_ty reg_fn
                         | Yes refl = Just reg_fn
  checkFunctions reg_ty frm_ty reg_fn
                         | No _ = Nothing
```

We may then parse the arguments according to the types specified by the handler function, and then apply the arguments to the handler function. Finally, we may run the handler function, ensuring that all updates made to the CGI state are propagated.

5. Extended Example: Message Board

In this section we consider a larger example—a message board application which allows users to register, log in, view and create threads, and list and create new posts in threads.

Firstly, we create a database schema in which to record information stored by the message board. We create three tables: `Users`, which contains a unique User ID, usernames and passwords; `Threads`, which contains a unique thread ID, a title, and the ID of the user who created the thread; and `Posts`, which contains a unique post ID, the ID of the thread to which each post belongs, the content of the post, and the ID of the user that created the post.

Secondly, we use a `GET` variable, `action`, to indicate which page of the message board should be displayed, and pattern-match on these to call the appropriate function which displays the page. Some pages, such as the page which shows all of the posts in a thread, require a second argument, `thread_id`.

5.1 Handling requests

The entry point to any CGI application is the `main` function. From here, we run the remainder of the program through a call to `runCGI`, which we initialise with empty initial environments for the `CGI`, `Session` and `SQLite` effects, so they may be used in further computations.

```
main : IO ()
main = do (runCGI [initCGIState,
                   InvalidSession, ()]
            handleRequest)
          return ()
```

We define a function, `handleRequest`, which firstly determines whether submitted form data must be handled, by checking whether a handler variable exists. If so, then the form handling routine is called, which executes the corresponding handler function as specified in Section 4.2. If not, then the `handleNonFormRequest` function is called, which inspects the `GET` variables in order to display the correct page.

```
handleRequest : CGIProg
     [SESSION (SessionRes SessionUninitialised),
      SQLITE ()] ()
handleRequest = do
  handler_set <- isHandlerSet
  if handler_set then do
    lift' (handleForm handlers)
    Effects.return ()
  else do
    action <- lift' (queryGetVar "action")
    thread_id <- lift' (queryGetVar "thread_id")
    handleNonFormRequest action (map strToInt thread_id)
```

5.2 Thread Creation

We create four forms: one to handle registration, one to handle logging in, one to handle the creation of new threads, and one to handle the creation of new posts. For example, the form used to

create a new thread contains elements for the title of the new thread and the content of the first post of the new thread:

```
newThreadForm : UserForm
newThreadForm = do
  addTextBox "Title" FormString Nothing
  addTextBox "Post Content" FormString Nothing
  useEffects [CgiEffect, SessionEffect, SqliteEffect]
  addSubmit handleNewThread handlers
```

This consists of two text boxes: one for the title of the thread, and one for the content of the first post. Both are of type `String`, as denoted by the `FormString` argument, and both have no default value. The handler function may make use of the `CGI`, `SESSION` and `SQLITE` effects, and the handler function is specified as `handleNewThread`. The `handlers` argument refers to the list of form handlers, and is of the following form:

```
handlers : HandlerList
handlers = [
 (handler args=[FormString, FormString],
          effects=[CgiEffect, SessionEffect, SqliteEffect],
          fn=handleRegisterForm,
          name="handleRegisterForm"),

 (handler args=[FormString, FormString],
          effects=[CgiEffect, SessionEffect, SqliteEffect],
          fn=handleNewThread,
          name="handleNewThread"),
     ...]
```

Creating a new thread (shown in Figure 10) requires a user to be logged in, so that the thread starter may be recorded in the database. In order to do this, we make use of the session handler. We define a function `withSession`, which attempts to retrieve the session associated with the current request, and if it exists, executes a function which is passed the associated session data. If not, then a failure function is called instead. Should the form handler function be called with invalid arguments, an error is shown.

```
handleNewThread :
  Maybe String -> Maybe String ->
  FormHandler [CGI (InitialisedCGI TaskRunning),
               SESSION (SessionRes SessionUninitialised),
               SQLITE ()]
handleNewThread (Just title) (Just content) = do
  withSession (addNewThread title content) notLoggedIn
  return ()
handleNewThread _ _ = do
  outputWithPreamble "<h1>Error</h1><br />There was
       an error posting your thread."
  return ()
```

Figure 10. Thread Creation

Once we have loaded the session data from the database, we then check whether the `UserID` variable is set, which demonstrates that a user has successfully logged into the system, and allows us to use the ID in subsequent computations. The database operation to insert the thread into the database is performed by `threadInsert`, shown in Figure 11. This uses a library function `executeInsert`, which abstracts over the low-level resource usage protocol, enabling for provably-correct database access without the excess boilerplate code. In addition, `executeInsert` returns the unique row ID of the last item which was inserted, which may be used in subsequent computations. In the case of the message board, we use this to associate the first post of the thread with the thread being inserted.

5.3 Listing Threads

Listing the threads in the database is achieved using `executeSelect`, which returns either a `ResultSet` or an error:

```
threadInsert : Int -> String -> String ->
               Eff IO [SQLITE ()] Bool
threadInsert uid title content = do
  let query = "INSERT INTO `Threads`
    (`UserID`, `Title`) VALUES (?, ?)"
  insert_res <- (executeInsert DB_NAME query
    [(1, DBInt uid), (2, DBText title)]
  case insert_res of
    Left err => return False
    Right thread_id => postInsert uid thread_id content
```

Figure 11. Thread Insertion

```
getThreads : Eff IO [SQLITE ()] (Either String ResultSet)
getThreads =
    executeSelect DB_NAME query [] collectThreadResults
 where query = "SELECT `ThreadID`, `Title`, `UserID`,
         `Username` FROM `Threads` NATURAL JOIN `Users`"
```

Once the result set has been retrieved, we may iterate through the results and output them to the page, including a link to a page which shows the posts associated with the thread. This is shown in Figure 12. Since we know the structure of the returned row from designing the query, we may pattern match on each returned row to make use of the returned values.

```
traverseThreads : ResultSet ->
  Eff IO [CGI (InitialisedCGI TaskRunning)] ()
traverseThreads [] = return ()
traverseThreads (x::xs) = do traverseRow x
                             traverseThreads xs
  where traverseRow : List DBVal ->
           Eff IO [CGI (InitialisedCGI TaskRunning)] ()
        traverseRow ((DBInt thread_id)::
                     (DBText title)::
                     (DBInt user_id)::
                     (DBText username)::[]) =
           (output $ "<tr><td>
            <a href=\"?action=showthread&thread_id=" ++
            (show thread_id) ++ "\">" ++
            title ++ "</a></td><td>" ++
            username ++ "</td></tr>")
        traverseRow _ = return ()
```

Figure 12. Thread Insertion

5.4 Authentication

Once a user submits the login form, the associated handler queries the database to ascertain whether a user with the given username and password exists through a call to the `authUser` function. This is shown in Figure 13. If so, then the session handler is invoked, and a session is initialised with the user ID retrieved from the database. The session ID is then set as a cookie using the CGI effect, so that it may be used in subsequent requests. Any failures, for example with creating a new session or querying the database, are reported to the user.

Implementations for the insertion and display of posts, alongside registration, follow the same structure.

Although we have described a relatively simple application, we have shown that through the use of embedded domain-specific languages, and particularly by encapsulating resource usage protocols in the types, we can write verified code that fails to compile should resources be incorrectly accessed. Additionally, we have used the form handling mechanism to simply handle the arguments passed by the user. Importantly, we have shown that dependent types can be used to increase confidence in an (albeit simplified) real-world application, without requiring developers to supply proofs or indeed work explicitly with dependent types.

```
handleLoginForm (Just name) (Just pwd) = do
  auth_res <- lift' (authUser name pwd)
  case auth_res of
    Right (Just uid) => do
      set_sess_res <- setSession uid
      if set_sess_res then do
        lift' (output $ "Welcome, " ++ name)
        return ()
      else do
        lift' (output "Could not set session")
         return ()
    Right Nothing => do
      lift' (output "Invalid username or password")
      return ()
    Left err => do
      lift' (output $ "Error: " ++ err)
      return ()
```

Figure 13. Thread Insertion

6. Related Work

Meijer [10] implemented a CGI library which was among the first libraries to handle web scripting monadically, and allows the user to implement application logic without having to consider the low-level details such as parsing in CGI data from the environment, or printing headers to the remote browser. The library also provides support for cookies and basic form handling.

Thiemann [18] adds the notion of a CGI Session for maintaining state, and provides more sophisticated form-handling methods. In particular, callbacks may be associated with submit buttons, with nameless representations for form inputs. Due to the unavailability of full dependent types in Haskell, however, this implementation does not statically verify the suitability of the callback function for the form inputs. Both implementations of the CGI library, being built upon monads, mean that the use of additional effects such as database access is achieved either through monad transformers or performing arbitrary IO operations. Both of these approaches are limited—the former does not scale well to multiple effects, and the latter allows for the introduction of errors by allowing the violation of resource usage protocols.

Plasmeijer and Achten [13] describe an alternative approach to type-safe form handling through the *i*nteractive *Data*, or iData abstraction. Instead of processing being triggered by form submission, as in the approach described in this paper, applications created in the iData toolkit are edit-driven. This means that upon a component being edited, a computation occurs, given the state of the current form. This is saved for future computations. Should a user enter invalid data, for example by entering text in a field designated for integers, the change will be reverted. This is demonstrated practically through the use of iData to implement a conference management system [14].

The concept of iData is taken further by the introduction of iTasks [15], which make use of a workflow system to allow multiple iData forms to interact with one another. This is achieved using high-level combinators which allow the implementation of concepts such as recursion, sequence and choice in a scalable fashion. These high-level abstractions are elegant, but the style and syntax differ substantially from a traditional web application. Our approach takes the concept of type-safe input handling and uses it in a more traditional fashion, whilst retaining the type-retention guarantees afforded by iData elements.

UrWeb [5] is a library built for the Ur language, which does not use full dependent types but does have an expressive type system with record types and type-level computation. By using these concepts, UrWeb may generate provably correct and unexploitable DOM code and SQL queries from records, without requiring developers to supply proofs. In contrast to using runtime code generation, which is prone to obscure code generation errors, UrWeb makes use of its static type system to guarantee that metaprograms—in this case, generated SQL and DOM code—must be correct and secure. Such ideas regarding the use of static checking of metaprogram generation will be extremely useful when considering an object-relational mapping system, which we hope to implement in the near future. It will also be interesting to see how such concepts may be applied with a yet more expressive type system involving full dependent types.

WebDSL [19] is a domain-specific language written primarily to introduce new abstractions which aim to reduce the amount of boilerplate code that must be written and maintained by developeres. WebDSL is built on top of Java, which often includes a large amount of redundant code such as accessor and mutator functions within entity classes. The Java Persistence API (JPA) [1] provides an object-relational mapping through the use of Java 5 annotations, which may then be used to construct database tables. These annotations soon become complex, however, and coupled with redundant boilerplate code, data model declarations may soon become unwieldy. Through the use of WebDSL, these data model declarations can be much more compactly declared, and elaborated into Java code by parsing the data-modelling DSL into an abstract syntax tree, applying rewrite rules, and pretty-printing. WebDSL also applies similar concepts to implement a *template system* for such objects, which allows the data to be used in code generation. We look to implement many of these ideas, but as effects within the IdrisWeb framework, as with the form construction effect.

7. Conclusions

Dependently-typed languages promise to support machine checkable program correctness proofs, but to date they have remained relatively unused for practical purposes. By using embedded domain-specific languages, we can abstract away some of the complexities of creating correctness proofs and provide expressive libraries, giving guarantees by the successful compilation of a program (assuming the use of specific enough types) without additional proofs.

Our framework provides several static guarantees. Data submitted by users is inherently unsafe and means systems are vulnerable to attacks such as SQL injection. This particular threat is ameliorated due to elements being associated with specific types during form construction. This immediately eliminates the possibilities of SQL injection attacks on non-string types. Since failures are handled transparently, no runtime errors are output to the browser, meaning that attackers may not use such information to aid attacks. Additionally, since checking is performed on the types of the form elements and the types of arguments accepted by the handler, it is impossible to associate a form with a handler incompatible with the submitted data.

Many external libraries also follow (unchecked or dynamically checked) resource usage protocols. Incorrect usage is however still possible, for example by forgetting to release acquired resources or failing to initialise a library correctly. By creating high-level bindings to these libraries, however, we may statically enforce these resource-usage protocols, ensuring that the libraries are used correctly. Whilst previous work has demonstrated that this is possible through the use of embedded DSLs [4] and dependent algebraic effects [3], this paper has provided more substantial examples of real-world applications.

In particular, the framework guarantees that it is not possible for a CGI application to produce an internal server error due to content being written to the remote host prior to headers. With regard to database access, we may statically guarantee that library calls are made in the correct order, and calls to retrieve rows of data are made only when more data is available. Additionally, by encoding desired

invariants within operation types, we may gain static guarantees about adherence to resource usage protocols and failure handling. Enforcing resource usage protocols also guards against common programmer errors, saving debugging time by identifying errors at compile time.

7.1 Further Work

We have shown that embedded domain-specific languages using dependent types and algebraic effects can be used to increase confidence in web applications by providing additional static guarantees about runtime behaviour, but much more can be done using the same approach.

There are many other applications which make use of specific resource usage protocols, for example popular libraries such as *libgcrypt*[2]. Applying a similar approach would allow for sensitive programs requiring cryptographic routines to be written using a language with full dependent types, in turn adding an extra layer of confidence in their security.

Whilst the use of CGI allows for experimenting with the use of dependent types in a real-world scenario such as web programming, there remain practical considerations about its scalability, as a separate process must be created for each individual request. We believe that the use of FastCGI may alleviate this, but ultimately, we would like to create a web server written in IDRIS, which would make more efficient usage of resources.

Since at this stage we have concentrated on the use of dependent types for enforcing resource usage protocols and type-safe form handling, we currently handle the generation of HTML in an unstructured manner. Future work will entail a DOM library to facilitate the generation and manipulation of HTML, in turn giving stronger guarantees about its correctness. Other planned features include a template system, allowing for web pages to be automatically generated from data, and an object-relational mapping system which will allow users to manipulate records which can then be automatically written to the database, instead of having to update tables manually via SQL queries.

Type providers, as originally implemented in F# [17], are an interesting method by which external data sources may be used to import external information, such that it may be used during compilation. In this way, it becomes possible to use the extra type information to statically ensure the validity of artefacts such as SQL queries and data structures. If data structures within the program do not conform to a given database schema, for example, then the program will not type-check. This has been implemented for IDRIS [6], exploiting the fact that types can be calculated by functions to avoid generating extra code in the type provider step. Dependent type providers additionally have stronger safety guarantees as they may not generate unchecked code, but at the same time this is matched by a decrease in expressiveness. Nonetheless, such techniques provide a promising mechanism to verify the semantic soundness of programs and we look to investigate their integration in further work.

Dependently-typed languages provide great promise for the construction of secure and correct programs. Through the use of embedded domain-specific languages, we hope that more developers may benefit from the extra guarantees afforded by dependent types, resulting in more stable, secure applications.

Acknowledgments

This work has been supported by the Scottish Informatics and Computer Science Alliance (SICSA) and the EPSRC. The authors would like to thank contributors to the IDRIS language, especially the authors of the original `Network.Cgi` and `SQLite` libraries.

[2] `http://directory.fsf.org/wiki/Libgcrypt`

References

[1] Heiko Böck. Java persistence api. In *The Definitive Guide to NetBeans Platform 7*, pages 315–320. Springer, 2011.

[2] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. . URL `http://journals.cambridge.org/article_S095679681300018X`.

[3] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 2013. To appear.

[4] Edwin Brady and Kevin Hammond. Resource-safe systems programming with embedded domain specific languages. In *Practical Aspects of Declarative Languages*, pages 242–257. Springer, 2012.

[5] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.

[6] David Raymond Christiansen. Dependent type providers. In *Workshop on Generic Programming (WGP '13)*, 2013.

[7] Lee Garber. Security, privacy, and policy roundup. *IEEE Security & Privacy*, 10(2):15–17, 2012. ISSN 1540-7993. .

[8] Imperva. Lessons Learned From the Yahoo! Hack. 2013. URL `http://www.imperva.com/download.asp?id=299`.

[9] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[10] Erik Meijer. Server side web scripting in haskell. *Journal of Functional Programming*, 10:1–18, 1 2000. ISSN 1469-7653. . URL `http://journals.cambridge.org/article_S0956796899003561`.

[11] OWASP. Cross-site Scripting (XSS). URL `https://www.owasp.org/index.php/Cross-site_scripting`.

[12] OWASP. SQL Injection, 2013. URL `https://www.owasp.org/index.php/SQL_injection`.

[13] Rinus Plasmeijer and Peter Achten. idata for the world wide web–programming interconnected web forms. In *Functional and Logic Programming*, pages 242–258. Springer, 2006.

[14] Rinus Plasmeijer and Peter Achten. A conference management system based on the idata toolkit. In *Implementation and Application of Functional Languages*, pages 108–125. Springer, 2007.

[15] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. itasks: executable specifications of interactive work flow systems for the web. *SIGPLAN Not*, 42:141–152, 2007.

[16] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *ESOP 09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 80—-94, 2009.

[17] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, et al. Strongly-typed language support for internet-scale information sources. Technical report, Technical Report. Microsoft Research, 2012.

[18] Peter Thiemann. Wash/cgi: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208. Springer, 2002.

[19] Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008.

[20] W3Techs. Usage of server-side programming languages for websites, July 2013. URL `http://w3techs.com/technologies/overview/programming_language/all`.