Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Core Language for Web Applications

Miguel Brazão Domingues (28063)

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Core Language for Web Applications

Miguel Brazão Domingues (28063)

Orientador: Prof. Doutor João Costa Seco

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Lisboa
(2010)

Acknowledgements

I'd like to thank ...

... my adviser Prof. João Seco for his support and guidance through the last year. Also Luísa Lourenço, Hugo T. Vieira, and Jorge Perez for extending the language with refinement types to express access control policies, which helped to improve the prototype.

... all those who helped in some way during all these years in FCT: friends, colleagues, and professors.

... my parents and all my family.

... Patuska for being *patuska*.
... Dadá for being crazy.
... Baco for bitting my ears.
... Trindade for spoiling my sleeves.
... Brutus for remembering me that at 18h it's time to eat.
... Dadá, Baco, Trindade and Brutus mom: Diana "Hunting Addicted" Brazão Domingues.
... my sister, who else would take care of all these dogs?

Last but not the least, I'd like to thank ..... for the support and patience showed till today.

THANK YOU ALL

PS: That space is for you to fill your name! Aren't you feeling important?! :)

Abstract

Web applications have a very high demand for rapid development and constant change. Several languages were created for this kind of applications, which are very flexible but many times trade the benefits of strongly-typed programming languages by untyped interpreted languages. With this kind of languages the interaction between different layers in a web application is usually developed using dialects and programming conventions with no real mechanical verifications between the client and server sides, and the SQL code within the application and the database.

We present a typed core language for web applications that integrates the typing of the interface definition, business logic, and database manipulation representing these interactions at a high abstract level. Using only one language, typed and with its own instructions to define the interface and the interaction with the database, becomes possible to make static checks. Thereby, avoiding execution errors caused by the usual heterogeneity among web applications. We also describe the implementation of a prototype with a highly flexible programming environment for our language that allows the application development and publishing tasks to be done through a web interface, interacting directly with the application and without loosing the integrity checks. This kind of development relies on an agile development methodology. Therefore, the modifications made to the application are made active using the dynamic reconfiguration mechanism, avoiding the recompilation of the application and system restart.

Keywords: Web application, database, business logic, interface, three layer integration, programming language, type system, dynamic reconfiguration

Resumo

As aplicações web estão sujeitas a constantes evoluções e a um ciclo de desenvolvimento rápido. Assim, foram criadas diversas linguagens para este tipo de aplicações, linguagens flexíveis que trocam os benefícios das linguagens com tipificação forte por linguagens interpretadas e não tipificadas. Com este tipo de linguagens a interacção entre as diferentes camadas de uma aplicação web é normalmente desenvolvida com recurso a protocolos e convenções, e não estão sujeitas a verificações estáticas entre o código do cliente e do servidor, e entre o código SQL da aplicação e a base de dados.

Neste trabalho apresenta-se uma linguagem tipificada para aplicações web que integra a tipificação da interface, lógica aplicacional e operações de manipulação de bases de dados. Com recurso a apenas uma linguagem com um nível de abstracção superior, tipificada e com instruções próprias para definir a interface e a interacção com a base de dados, torna-se possível efectuar verificações estáticas, evitando erros comuns causados pela heterogeneidade existente nas aplicações web. Também se apresenta um protótipo de um ambiente de desenvolvimento flexível que permite realizar tarefas de desenvolvimento e publicação de aplicações, através de uma interface web, interagindo directamente com a aplicação. Este tipo de desenvolvimento de aplicações baseia-se num desenvolvimento ágil, onde modificações efectuadas na aplicação são tornadas activas utilizando um mecanismo de reconfiguração dinâmica, evitando assim que seja necessário recompilar a aplicação e reiniciar o sistema.

Palavras-chave: Aplicação web, base de dados, lógica aplicacional, interface, integração de três camadas, linguagem de programação, sistema de tipos, reconfiguração dinâmica

Contents

List of Figures

Listings

Introduction

This MSc work aims at designing and implementing a core programming language for web applications. We design a typed core language for web applications that integrates the typing of interface definition, business logic and database manipulation, and, at the same time, represents the interactions between layers at a higher level of abstraction. We also describe the implementation of a highly flexible development environment where the developers act directly over the installed application using a web browser.

1.1 Motivation

In the beginning of the World Wide Web in 1989 [8], all web sites were composed just by static content, i.e., web servers containing several files that could only be modified on the servers them self. In 1993, the Common Gateway Interface (CGI) [28] revolutionized the way Internet was viewed till then. It made possible to create web sites with dynamic content by executing small programs on the server that could generate an HTML page for each request made to the server. In the early times, web applications were structured in a simple *client – server* architecture but later, with the use of databases to store application data, web applications architecture evolved to a three layer architecture that divides applications into client interface, business logic, and

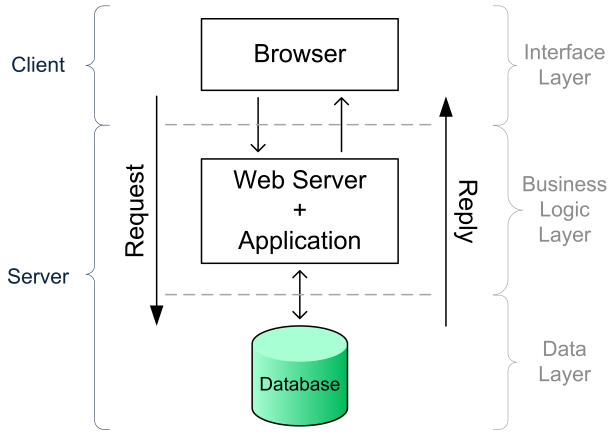


Figure 1.1: Three Layer Architecture [32]

data manipulation layers [32].

Figure 1.1 shows the interaction between client and server in a three layer architecture and is processed as follows:

1. The user requests a resource via a URL through the browser;

2. The web server analyzes the request and executes an application and, if needed, contacts the data layer to collect data;

3. After gathering the data, the server returns the requested resource, often an HTML page.

In addition to code execution on the server side, it is also possible to execute code on the client side (client-side scripting). Java Applets, JavaScript and Flash are among most common technologies used as client-side scripting. More recently the term Asynchronous JavaScript, and XML (AJAX) gained some reputation, through the combination of several techniques in AJAX it is possible to update a table without needing to reload the entire page. The data required to fill the table is retrieved from the server asynchronously in the background without interfering with the look and behavior of the current page, after receiving the data, the browser updates the table using JavaScript. Applications such as GMail or Google Maps are some examples of heavy AJAX usage.

Making web sites more attractive, interactive and innovative require web applications to integrate several languages (e.g., HTML, JavaScript, PHP, SQL). Since most languages do not support the integration between the several web application layers, static checks between layers are insufficient or do not exist at all. This causes errors to be detected only during runtime, errors that could be easily detected if the application were developed using a language with static checks. Using several languages

also makes the development and maintenance tasks more complex due to the need to integrate several languages and, although the user gains in usability, the developer work gets harder. Moreover, web applications are usually developed in heterogeneous environments where different languages interact using dialects and conventions with no real mechanical verifications of the connection code between client and server sides, and the SQL code issued within the application and the actual database model [15].

A very high demand for rapid development and constant change in this kind of applications gave rise to a series of flexible languages that many times trade the benefits of strongly-typed main stream programming languages (e.g., Java, C#) for the advantages of interpreted languages (e.g., PHP, ASP, Ruby). Some of these main-stream interpreted languages provide scaffolding frameworks and programming patterns to improve developers' productivity, and overcome the most common programming errors [1, 5, 12]. Other approaches introduce extensions to standard typed programming languages that integrate database manipulation in the typing process [3, 10, 18, 31] or by designing Domain Specific Languages (DSL) that seek to eliminate programming errors by construction [4, 13, 35]. Thus, we consider that is advantageous to use a programming language that allows to integrate all three layers of a web architecture in a simple and effective manner and at the same time support static checks.

The motivation behind this work comes from languages like the one developed by OutSystems [4]. The Agile Platform from OutSystems is based on a single programming language and enables the integration of the three existing layers in a web architecture. Through this platform the application development process, usually complex and time consuming, is turned into a process with higher productivity levels. Tasks like publishing a web application are executed with a single click, therefore in a smaller amount of time it is possible to develop an entire web application.

This master thesis is also part of a collaboration involving CITI[1] and OutSystems, the Certified Interfaces project (Carnegie Mellon – Portugal). The objective of this research is to study properties related to data security and access control [9, 26, 33] and related coordination of the several interacting parts [34]. However, the language from OutSystems is too complex to support this kind of formal studies, so in order to be possible to perform this kind of studies a smaller and simpler language is required.

1.2 Context

In the context of this work it is important to have a model language for the OutSystems platform enabling studies about the language. Through a language similar to the one in

[1] Center for Informatics and Information Technology – DI/FCT-UNL

the platform developed by OutSystems it is possible to study, in a simple and effective way, several problems in the design of a language that integrates the three layers from a common web application architecture.

The language designed in this work aims at offering an integrated programming environment allowing to define the interface, business logic, and database manipulation operations. Computations are usually specified with general purpose languages and database operations with specialized query languages, and typically different layers interact through dialects and programming conventions, and communication code is not subject to effective mechanical verifications and is highly error prone. Our language uses a higher level of abstraction, in comparison to general purpose languages, avoiding the several mismatches between layers by using static verifications within all application code, including the communication code between layers.

Rising the level of abstraction allows for checking the basic safety of programs and elimination of many programming errors. Our language aims at potentiating the verification of other more sophisticated properties, in particular we refer to properties related to data security and access control [9, 26, 33], and related to the coordination of several interacting parts in distributed systems [34].

1.3 Proposed Solution

We present a typed core language for web applications that integrates the typing of interface definition, business logic and database manipulation operations. Interactions between layers are represented at a higher level of abstraction providing basic safety of properties and elimination of many common programming errors.

An application is divided in three kinds of programming elements: entities (*Model*), screens (*View*), and actions (*Controller*), similar to the Model-View-Controller pattern [27]. Each application is contained inside a module, that works like a namespace allowing to have several running applications in the same system without having conflicts among them.

Entities are containers of structured persistent data implemented as database tables. Entity attributes are defined using types provided by our language, avoiding type mismatches between layers, in particular between the database layer and business logic layer. Also, by using the same type system in all layers is possible to perform static checks on an entire application code and the communication code between layers. Our language has native support for querying entities, this query language mimics a subset of the standard query language (SQL).

Screens allow to define the user interface, i.e., the web pages that are sent to the

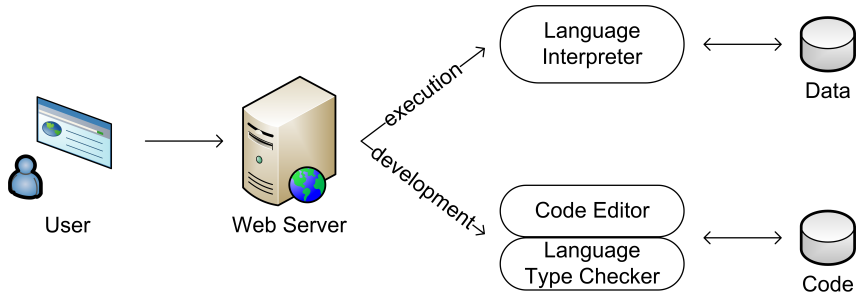


Figure 1.2: Prototype Interaction

user browser. User interface definition is done using abstractions over an interface definition language that mimics a subset of basic HTML elements (e.g., text fields, buttons, div containers). Screens may be parameterized and some of the user interface expressions may contain general purpose expressions to be executed back at the server.

The core of the application is defined by actions that can also be parameterized. Actions are abstractions over general purpose expressions comprising operations over entities, screens, and other values allowing to define the application business logic.

Our implementation consists in a web server divided in two parts as depicted in Figure 1.2: execution mode web server and a web based development environment. The first part consists in an application browser where users can request web pages and interact with the application through links and forms. Web based development environment allows to create and modify existing applications. Through the web based environment users can save new versions of each definition (entities, screens, or actions) that are then submitted to the language type checker. When a new definition is submitted and considered sound by the type checker it becomes immediately active, i.e., the dynamic reconfiguration mechanism loads the new application definition and makes it available in execution mode.

1.4 Contributions

Our solution aims at offering an integrated programming language to define interface, business logic, and database manipulation operations. This work also aims at developing a prototype for the language that allows us to define an entire application using a web based development environment and also interact with the applications created.

Integration. The integration of the three layers from a common web architecture is one of the most important aspects during web applications development. With a programming language that integrates all three layers, thus not requiring multiple languages for a single application, we can provide static checks not only in each layer, but

also in the communication code between layers, therefore eliminating many common programming errors.

Runtime Support System. Our implementation aims to provide a simple and effective workbench for studies, so we present a web based development environment close to a Wiki system. Through this web based environment users can create and modify applications, and then publish them with a single click. Upon publishing an application, the code is submitted through the static checks available in the language, and only if the entire code is considered sound by the type checker, it will be published.

Dynamic Reconfiguration. The dynamic reconfiguration mechanism allows to modify an existing application without requiring to restart the web server or recompile the application code. When the application code is submitted, if the entire code passes through the static checks, the old running application is updated to the new definition. During this update, entities are updated to match the new definitions, and new actions or screens definitions are loaded.

Further Studies. One of the goals underlying this work was to provide a working prototype for a language that could be easily extended to allow further studies, like data security and access control properties. This has been achieved since the language is already being extended to demonstrate security related property checking by means of refinement types [9, 21].

1.5 Document Structure

In the following Chapter 2 we start by describing some practical examples using our language. Then we present our language by detailing the language syntax, semantic rules, and type system rules. Chapter 3 describes the implementation done in our prototype. This description includes the prototype architecture, and both modes where users can interact directly with applications and where users interact with the web based development environment. In Chapter 4 we present some frameworks that target at web applications by providing scaffolding features to increase developers productivity. We also handle the impedance mismatch between programming languages and databases integration by describing some extensions to general purpose languages that include the typing of database operations. At the end of Chapter 4 we describe some frameworks that use domain specific languages to provide program safety by construction. Finally, in Chapter 5 we present the final remarks for this work and describe some features that can be added to our language as future work.

2

Core Language for Web Applications

We present a typed core language for web applications that integrates the typing of interface definition, business logic, and database manipulation operations. Interactions between layers are represented at a higher level of abstraction providing basic safety of programs and elimination of many programming errors. Our core language has three programming elements: entities, screens, and actions. Entities are containers of structured persistent data implemented in database tables. Operations over entities mimic a subset of the standard query language (SQL). Screens are abstractions over a user interface definition language whose values are web pages. Screens may be parameterized and some of the user interface expressions may contain general purpose expressions to be executed back at the server. Actions are abstractions over general purpose expressions comprising operations over entities, screens, and other values.

We explain three web applications examples using our language (Section 2.1) and also present the language syntax (Section 2.2), semantics (Section 2.3), and type system (Section 2.4). This formal presentation prepares ground for extending the language with richer type languages [9] and introducing soundness proofs following standard techniques, however this deeper work lies out of the scope of this work.

```
def entity Person {
  id: Id,
  name: String,
  phone: String
}

def screen directory {
  iterator (row in (from (p in Person) select p)) {
    label "Name: " + row.name; br;
    label "Phone: " + row.phone; br; br
  };
  label "Name: "; textfield name; br;
  label "Phone: "; textfield phone; br;
  button "Add" to addPerson(name, phone)
}

def action addPerson(nameIn: String, phoneIn: String): Block {
  insert {
    name = nameIn,
    phone = phoneIn
  } in Person;
  directory()
}
```

Listing 2.1: Phone Book Example

2.1 Examples

In this section we describe three applications designed with our language. The first example shows a simple phone book (Section 2.1.1) to store phone numbers associated with names. Then a similar example for a simple message blog (Section 2.1.2) that allows anyone to post text messages. The last example describes a photo album (Section 2.1.3) for several users where new photos can be added using the photo album owner's password.

2.1.1 Phone Book

Consider the code in Listing 2.1 that implements a phone book directory. It defines an entity called Person containing names (name) and phone numbers (phone), a screen directory and an action addPerson.

The screen directory is defined around a **from** expression (written in a syntax similar to LINQ [10]) that fetches all values stored in the entity Person. The **iterator** in the screen definition expands the query result into a sequence of web page blocks computed by iterating over the elements of the query result. The screen also contains **textfield** elements, which bind the user input elements with the local names name and

Figure 2.1: Phone Book directory Screen

phone. There is also a **button** element that calls a server action named addPerson using as arguments the values in the text fields associated to the local names name and phone. Figure 2.1 displays the resulting HTML screen page, already with two records in the entity. The header and footer page layout comes directly from the web server, the middle section contains the result from the screen directory execution.

When the "Add" button is pressed, the action addPerson is executed at the server adding a new row to the entity Person. The values for this new row are obtained from the parameters in the action addPerson that were sent in the request made in the previous screen. After the insertion of the new row, the screen directory is rendered again.

2.1.2 Simple Blog

This Simple Blog example allows any user to post messages without any kind of verifications. However, security code could be added, e.g., through a User entity and forms to register and login, we avoided to include such mechanisms in order to keep the example simple. Listing 2.2 code fragment displays entity Message where messages are stored. Each message has a title, author and a body (text).

The list screen presented in Listing 2.3 lists all current messages. The **iterator** expression iterates a **from** query result that selects all messages from the entity Message. For each row the message title (row.title) and author (row.author) is presented along side with a View link to the screen view (Listing 2.5). After displaying all message titles, authors and links, a Write Message form is displayed. The label Write Message is formatted using a **div** block and a CSS class named title (Listing 2.4). At the screen bottom three

```
def entity Message {
  id: Id,
  title: String,
  author: String,
  text: String
}
```

Listing 2.2: Simple Blog Entity

```
def screen list {
  div title {
    label "Message List"
  };
  iterator(row in (from (m in Message) select m)) {
    label row.title + " (" + row.author + ") ";
    link {
      label "View"
    } to view(row.id);
    br
  };
  br; br;
  div title {
    label "Write Message"
  };
  label "Title: "; textfield title; br;
  label "Author: "; textfield author; br;
  label "Text: "; textfield text; br;
  button "Add" to saveMessage(title, author, text)
}
```

Listing 2.3: Simple Blog list Screen

textfield and a **button** are displayed. These elements allow to add a new message by calling the server action saveMessage and sending the text fields input values by argument. Figure 2.2 displays the result of rendering the screen list already with two messages in the entity.

Listing 2.5 contains the view screen definition. In this definition, through a **from** query that selects a message using the message identifier (msgId) in a **where** clause, an **iterator** displays the message details. Although the **iterator** is normally used to display a set of results (more than one), in this case, the **iterator** is here used to iterate only one message, selected by the entity primary key id. Inside the **iterator** block, after displaying the message title and author formatted with the **div** block, the message text is displayed. Then a **link** to the list screen is displayed allowing to go back to the main screen list. Figure 2.3 shows an example of the view screen web page.

The final part of the list screen (Listing 2.3) displays a form to add messages. When the **button** from that screen is clicked, the action saveMessage (Listing 2.6) is executed

```
.title {
  font-size: 24pt;
  font-weight: bold;
  color: #515151;
  border-bottom: 1px dashed #9A9A9A;
}
```

Listing 2.4: Simple Blog and Photo Album title CSS

Figure 2.2: Simple Blog list Screen

```
def screen view(msgId: Int) {
  iterator(msg in (from (m in Message) where m.id == msgId select m)){
    div title {
      label "'" + msg.title + "' by " + msg.author
    }; br;
    label msg.text
  };
  br; br;
  link {
    label "Message List"
  } to list()
}
```

Listing 2.5: Simple Blog view Screen

in the server with the arguments from the input text fields values. This saveMessage action only adds a new message to entity Message and then displays the full message list, by rendering the screen list.

Figure 2.3: Simple Blog view Screen

```
def action saveMessage(titleIn: String, authorIn: String, textIn: String): Block {
  insert {
    title = titleIn,
    author = authorIn,
    text = textIn
  } in Message;
  list()
}
```

Listing 2.6: Simple Blog saveMessage Action

2.1.3 Photo Album

Photo Album application allows users to browse photos from all users, and also allows each user to add new photos to their own album. The code fragment in Listing 2.7 defines two entities (User and Photos). User entity contains details for each user, it includes the user name, password and email address. The second entity, Photos stores photos details for all users where each photo is identified by its owner (user), a title and the photo url.

The userList screen (Listing 2.8) displays a list of all users in the system. First, the label User List is formated with a **div** element and the CSS class title (Listing 2.4). Using an **iterator** expression, the query that selects all user's names is expanded. In each expanded row, a link to each user's album is created, i.e., a link to the viewUser screen. After displaying the user list though links, a simple register section is presented, containing three **textfield** elements and a **button** element to call the register action at the server. Figure 2.4 shows the resulting screen, already with two users in the system.

The register action (Listing 2.9) adds a new row to the User entity based on the parameters values, and after the row is inserted the userList screen is shown again.

```
def entity User {
  id: Id,
  username: String,
  password: String,
  email: String
}

def entity Photos {
  id: Id,
  user: String,
  title: String,
  url: String
}
```

Listing 2.7: Photo Album Entities

```
def screen userList {
  div title {
    label "User List"
  };
  iterator ( row in (from (u in User) select {uname = u.username})) {
    link {
      label row.uname
    } to viewUser(row.uname); br
  }; br;

  div title {
    label "Register"
  };
  label "Username: "; textfield username; br;
  label "Password: "; textfield password; br;
  label "E-mail: "; textfield email; br;
  button "Register" to register(username, password, email)
}
```

Listing 2.8: Photo Album userList Screen

```
def action register(uname: String, pwd: String, em: String):Block {
  insert {
    username = uname,
    password = pwd,
    email = em
  } in User;
  userList()
}
```

Listing 2.9: Photo Album register Action

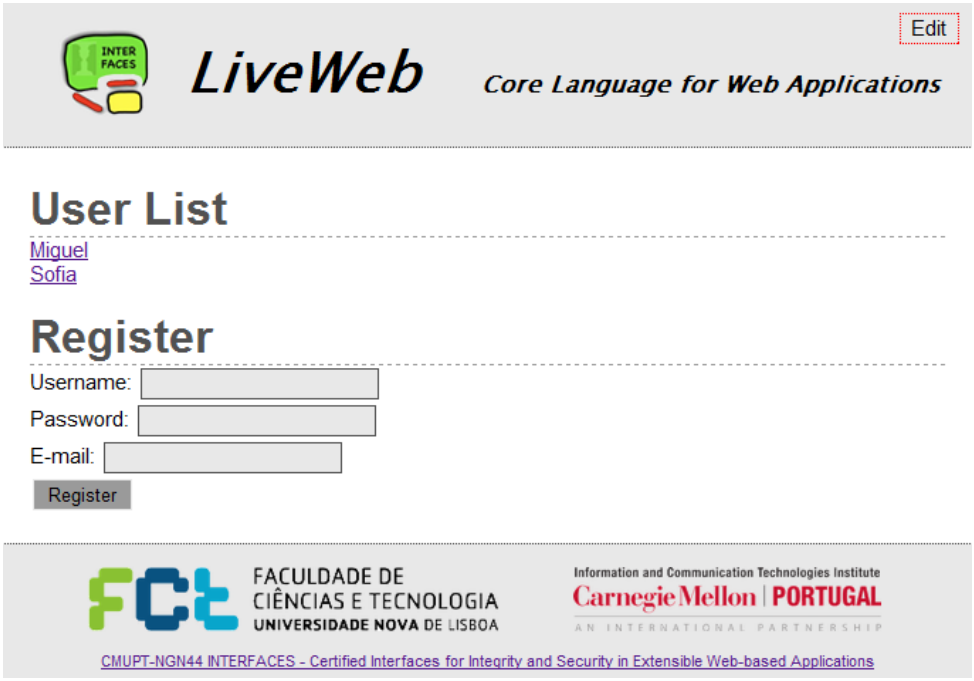


Figure 2.4: Photo Album userList Screen

```
def screen viewUser(uname: String) {
  div title {
    label uname + "'s Photos"
  };
  iterator (photo in (from (p in Photos)
                       where p.user == uname
                       select {title = p.title, url = p.url})) {
    label photo.title; br;
    image photo.url; br; br
  }; br; br;

  div title {
    label "Add Photo"
  };
  label "Password: "; textfield password; br;
  label "Photo Title: "; textfield title; br;
  label "Photo URL: "; textfield photo; br;
  button "Add" to addPhoto(uname, password, title, photo); br; br;

  link {
    label "Back to User List"
  } to userList()
}
```

Listing 2.10: Photo Album viewUser Screen

```
def action addPhoto(uname: String, pwd: String, title: String, url: String): Block {
  let q = from (u in User)
        where u.username == uname and u.password == pwd
        select u in
    (if (count(q) == 1) then {
          insert {
        user = uname,
        title = title,
            url = url
          } in Photos
    } else {
      false
    });
  viewUser(uname)
}
```

Listing 2.11: Photo Album addPhoto Action

After clicking in a user name link in the userList screen, the viewUser (Listing 2.10) screen is displayed. This screen executes a **from** query selecting all photos for a given user (parameter uname) using a **where** clause. The query result is iterated and for each value, the photo title and **image** are displayed. After displaying all photos, an Add Photo form is displayed. Through this form it is possible to add new photos, with the user's password and the new photo title and URL. The Add **button** calls the server action addPhoto (Listing 2.11) with text fields input values as arguments. Figure 2.5 displays the viewUser screen for the user *Miguel* with one photo.

When the button in the userList screen is clicked, the addPhoto action (Listing 2.11) is executed at the server. The action checks if the username and password are valid, if the query result contains exactly one row (case where the user details are correct), then the new photo is inserted in the entity Photos. After that, the viewUser screen for the given user name uname is displayed again.

2.2 Syntax

Our language comprises three language fragments to define entities, web page blocks, and expressions according to the syntax in Figure 2.6. The definition of language values is in Figure 2.7 and the definition of the type language is in Figure 2.8.

An application is composed by a set of definitions for entities, actions, and screens. Entities are containers of structured persistent data implemented in database tables. Each defined entity is identified by an entity name (t), an auto-increment integer primary key and a list of typed attributes. Operations over entities mimic a subset of

Figure 2.5: Photo Album viewUser Screen

the standard database query language (SQL). The core of the application, i.e., business logic, is defined by actions which are abstractions over general purpose expressions comprising operations over entities, screens, and other values. Each action can be parameterized and must have a declared return type. Application interface is defined in pre-designed screens, which are abstractions over a user interface definition language whose values are web page blocks. This approach allows us to define user interfaces in a clean separate way, which is distinctive from other languages that mix both concepts in the same language.

Basic types used in entity attributes are integers (Int), strings ($String$), and booleans ($Bool$). We also define, for each entity an identifier type ($entity$.**Id**) whose internal representation is a foreign integer key, i.e., another entity integer primary key. In addition to basic types, our language has structure, list, and block types. Structures are labeled product types that describe instances of entities (or rows of database tables) and list types describe homogeneous sequence of values. The $Block$ type represents web page blocks, i.e., screen elements.

Available expressions comprise all common binary and unary operators. It also

$t, a, s \in \mathcal{I}$, set of definition identifiers
$id \in \mathcal{V}$, set of variable identifiers
$label \in \mathcal{L}$, set of labels

$\mathcal{A}$	::=	$\overline{\mathcal{D}}$	(Application)
$\mathcal{D}$	::=		(Definitions)
		def entity t { $label$: **Id**, $\overline{label : \mathcal{BT}}$ }	(Entities)
	|	**def action** a ($\overline{x : \mathcal{T}}$) : $\mathcal{T}$ { e }	(Actions)
	|	**def screen** s ($\overline{x : \mathcal{T}}$) { b }	(Screens)
$\mathcal{O}$	::=		(Binary Operators)
		;	
	|	**and** | **or**	
	|	$+ \mid - \mid * \mid /$	
	|	$< \mid > \mid <= \mid >= \mid == \mid !=$	
$\mathcal{U}$	::=	**not** $\mid -$	(Unary Operators)
e	::=		(Expressions)
		$e\ \mathcal{O}\ e$	(Binary Operation)
	|	$\mathcal{U}\ e$	(Unary Operation)
	|	v	(Values)
	|	**let** $x = e$ **in** e	(Variable Declaration)
	|	**if** e **then** e **else** e	(Condition)
	|	a ($\overline{e}$)	(Action/Screen Call)
	|	[$\overline{e}$]	(List)
	|	**foreach** x **in** e **do** e	(List Iterator)
	|	{ $\overline{label = e}$ }	(Structure)
	|	$e.label$	(Structure Field)
	|	**insert** e **in** t	(Entity Insert)
	|	**update** x **in** t **with** e **where** e	(Entity Update)
	|	**from** ($\overline{x$ **in** $t}$) **where** e **select** e	(Entity Select)
	|	**count** (e) | **max** (e) | **min** (e)	(Aggregation Functions)
b	::=		(Web Page Blocks)
		$\overline{b}$	(Sequence of Blocks)
	|	**br**	(Line Break)
	|	**label** e	(Label)
	|	**div** $label$ { b }	(Div)
	|	**image** e	(Image)
	|	**link** { b } **to** e	(Link)
	|	**iterator** (x **in** e) { b }	(Iterator)
	|	**textfield** x **with** e	(Text Field)
	|	**button** e **to** e	(Button)

Figure 2.6: Syntax – Definitions, Expressions and Web Page Blocks

$$
\begin{array}{lcll}
w & ::= & & \text{(Web Page Blocks Values)} \\
 & & \overline{w} \mid \textbf{br} \mid \textbf{label}\ v \mid \textbf{div}\ label\ \{\ w\ \} \mid \textbf{image}\ v & \\
 & \mid & \textbf{link}\ \{\ w\ \}\ \textbf{to}\ e \mid \textbf{iterator}\ (\ x\ \textbf{in}\ v\ )\ \{\ w\ \} & \\
 & \mid & \textbf{textfield}\ x\ \textbf{with}\ v \mid \textbf{button}\ v\ \textbf{to}\ e & \\
 & & & \\
v & ::= & string \mid int \mid \textbf{false} \mid \textbf{true} \mid id \mid [\overline{v}] \mid \{\overline{l = v}\} & \text{(Expressions Values)}
\end{array}
$$

Figure 2.7: Syntax – Values

$$
\begin{array}{lcll}
\mathcal{BT} & ::= & & \text{(Basic Types)} \\
 & & String & \text{(Strings)} \\
 & \mid & Int & \text{(Integers)} \\
 & \mid & Bool & \text{(Booleans)} \\
 & \mid & entity.\textbf{Id} & \text{(Entity Identifier)} \\
 & & & \\
\mathcal{T} & ::= & & \text{(Types)} \\
 & & \mathcal{BT} & \text{(Basic Type)} \\
 & \mid & Block & \text{(Web Page Block)} \\
 & \mid & \{\ \overline{label\ :\ \mathcal{T}}\ \} & \text{(Structure)} \\
 & \mid & List\langle \mathcal{T} \rangle & \text{(List)}
\end{array}
$$

Figure 2.8: Syntax – Types

contains the standard conditional and function call expressions, and a functional fragment that contains **let** declarations. The language works with primitive list values for which we define a list constructor by evaluating a list of expressions and a **foreach** expression that iterates a given list of values, and executes its inner expression for each value producing a list of return values. Structures are also primitive values in our language, so we have the standard structure constructor and field access expressions.

Database manipulation adds an imperative flavor to the language and is incorporated in the expression language by the **insert**, **update**, and **from** expressions that mimic a subset of the standard SQL queries in a syntax similar to LINQ [10]. The **from** expressions fetch a list of rows from a set of entities, corresponding to a relational join, and a conditional **where** expression. The **insert** expression adds a new row to a given entity, while the **update** expression replaces values of fields in one or more rows in the entity. The available aggregate functions **count**, **min**, and **max** take an expression and applies the function to that expression value. Although these operations usually correspond to primitive database operations, it is not yet supported in our core language.

A web page is composed by a non-empty set of web page blocks where each kind of block represents a different sort of HTML element. The web page block language

comprises line breaks (**br**), text labels (**label**), named **div** blocks, which are used to give structure and custom look to blocks based on classes defined in the Cascading Style Sheet (CSS). There are also blocks to introduce images (**image**) and links (**link**) for a given URL. The **iterator** block expands a list of values to a set of blocks for each value in the list. Input elements (**textfield**) declare local variable names that can be used in expressions that pass the control flow from the browser back to the web server through actuator elements (**button**). Web page blocks may contain expressions and actually a web page is defined by web page blocks containing values or delayed expressions in the case of buttons.

2.3 Semantics

We define our language semantics in a big-step rule system where the successful evaluation of an expression e, with relation to a program state S and a program definition P, is given by a valid judgment of the form:

$$S; e \Downarrow S'; v$$

Where S and S' are the program states before and after the evaluation. The program state is a mapping between entity names and mutable collections of structured values. We write $S(t) = [v_1, \cdots, v_n]$ to express that the entity t has n rows, v_1 to v_n without any specific order. The expression e in the judgment is the expression to be evaluated in the context of program P. We say that the evaluation of the expression e in the state S results in a value v and a changed state S'. We also write $P(x) = y$ to fetch the definition y of name x in program P and, $e\{w/x\}$ to mean the substitution of variable x by value w in the expression e.

Binary operators semantic rules perform a left to right evaluation. First the left-hand side expression e_1 evaluates to a value v_1 and then the right-hand side expression e_2 evaluates to v_2. The resulting value corresponds to the operation denoted by the binary ϕ operator. Sequence operator (;) has a special evaluation, since the resulting value corresponds only to the value associated to the right-hand side expression e_2. However, both expressions, e_1 and e_2 are evaluated, also from left to right.

$$\phi \in \{\textbf{and}, \textbf{or}, ==, !=, >, >=, <, <=, +, -, *, /\}$$

$$\frac{S; e_1 \Downarrow S'; v_1 \qquad S'; e_2 \Downarrow S''; v_2}{S; e_1\ \phi\ e_2 \Downarrow S''; v_1\ \phi\ v_2} \qquad \frac{S; e_1 \Downarrow S'; v_1 \qquad S'; e_2 \Downarrow S''; v_2}{S; (e_1; e_2) \Downarrow S''; v_2} \quad \text{(Binary Operation)}$$

Unary operators semantic rule is straight forward: expression e evaluates to value

v and then the unary operator θ is applied.

$$\theta \in \{\textbf{not}, -\} \qquad \frac{S; e \Downarrow S'; v}{S; \theta\ e \Downarrow S'; \theta\ v} \qquad \text{(Unary Operation)}$$

Variable declaration allows to associate a value with an identifier. Expression e_1 evaluates to a value v_1 and then the second expression e_2 is evaluated after replacing all occurrences of label x by the value v. This last evaluation returns a value v_2, the same value for the whole **let** expression.

$$\frac{S; e_1 \Downarrow S'; v_1 \qquad S'; e_2\{v_1/x\} \Downarrow S''; v_2}{S; \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow S''; v_2} \qquad \text{(Variable Declaration)}$$

The conditional expression **if** takes three expressions, the condition e_c, the **then** expression e_1, and the **else** expression e_2. Expression e_c must evaluate to **true** or **false**. If e_c evaluates to **true**, expression e_1 is evaluated to the value v. Otherwise, if e_c evaluates to **false**, expression e_2 is evaluated to the value v. In either cases, only one of the branch expressions, e_1 or e_2, is evaluated.

$$\frac{S; e_c \Downarrow S'; \textbf{true} \qquad S'; e_1 \Downarrow S''; v}{S; \textbf{if}\ e_c\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow S''; v} \qquad \frac{S; e_c \Downarrow S'; \textbf{false} \qquad S'; e_2 \Downarrow S''; v}{S; \textbf{if}\ e_c\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow S''; v} \qquad \text{(Condition)}$$

The rule for action call follows a call-by-value strategy, where expressions $\overline{f}$ used as arguments are evaluated to obtain values $\overline{g}$. The body expression e is evaluated to value v with the parameters x_i replaced by the values g_i.

$$\frac{P(a) = a(\overline{x})\{e\} \quad S_i; f_i \Downarrow S_{i+1}; g_i \qquad S_{n+1}; e\{\overline{g}/\overline{x}\} \Downarrow S'; v \quad i = 1, \cdots, n}{S_1; a\ (\overline{f}) \Downarrow S'; v} \qquad \text{(Action Call)}$$

The semantics for the screen call is similar to action call. The difference remains in the definition body, the screen body is a block b that is evaluated to a web page block value w.

$$\frac{P(s) = s(\overline{x})\{b\} \quad S_i; f_i \Downarrow S_{i+1}; g_i \qquad S_{n+1}; b\{\overline{g}/\overline{x}\} \Downarrow S'; w \quad i = 1, \cdots, n}{S_1; s\ (\overline{f}) \Downarrow S'; w} \qquad \text{(Screen Call)}$$

The list constructor takes a set of expressions $\overline{e}$ and evaluates each expression e_i to a value v_i, and as result we obtain a sequence of values in the same order as the expressions.

$$\frac{S_i; e_i \Downarrow S_{i+1}; v_i \qquad i = 1, \cdots, n}{S_1; [\ \overline{e}\ ] \Downarrow S_{n+1}; [\ \overline{v}\ ]} \qquad \text{(List)}$$

The rule for the **foreach** expression evaluates a given expression e to a list of values and then repeatedly evaluates the expression f for each element in the list replacing all

occurrences of x by the value v_i from the list.

$$\frac{S; e \Downarrow S_1; [v_1, \cdots, v_n] \qquad S_i; f\{v_i/x\} \Downarrow S_{i+1}; w_i \qquad i = 1, \cdots, n}{S; \textbf{foreach } x \textbf{ in } e \textbf{ do } f \Downarrow S_{n+1}; [w_1, \cdots, w_n]} \quad \text{(List Iterator)}$$

A structure is created based on an association between a set of labels $\bar{l}$ and a set expressions $\bar{e}$, where each label l_i is associated to the expression e_i. Each expression e_i evaluates to a value v_i and is associated to the same label l_i that the expression e_i was associated. The result is a set of values, each associated with a single label.

$$\frac{S_i; e_i \Downarrow S_{i+1}; v_i \qquad i = 1, \cdots, n}{S_1; \{\ \overline{l = e}\ \} \Downarrow S_{n+1}; \{\ \overline{l = v}\ \}} \quad \text{(Structure)}$$

To access a structure field, first expression e is evaluated to a structure value, and then the value associated with the specified $label$ is returned.

$$\frac{S; e \Downarrow S'; \{\ \cdots, label = v, \cdots\ \}}{S; e.label \Downarrow S'; v} \quad \text{(Structure Field)}$$

The rule for **insert** evaluates an expression e to a structure and adds a new row to an existing entity t. The resulting value is **true** if the new value was successfully appended. If there is a conflict between the new value v to be inserted and any existing value v_i, the entity is not modified, i.e., $S'(t) = S(t)$ and the **insert** expression returns **false**.

$$\frac{S(t) = [v_1, \cdots, v_n] \quad S; e \Downarrow S'; v \quad S'(t) = [v_1, \cdots v_n, v] \quad \forall i, v \neq v_i}{S; \textbf{insert } e \textbf{ in } t \Downarrow S'; \textbf{true}} \quad \text{(Entity Insert – 1)}$$

$$\frac{S(t) = [v_1, \cdots, v_n] \qquad S; e \Downarrow S'; v \qquad S'(t) = S(t) \qquad \exists i, v = v_i}{S; \textbf{insert } e \textbf{ in } t \Downarrow S'; \textbf{false}} \quad \text{(Entity Insert – 2)}$$

The **update** expression modifies a set of values in the entity t. For each existing value in the entity that the condition f holds, expression e is evaluated, modifying the existing value v_j and obtaining a new value w_j. For the values v_k that the condition f is **false**, that value remains unmodified. The **update** expression result is the set of values that were modified plus the values that were not modified. The number of records in the entity before and after the whole expression is the same.

$$\frac{\begin{array}{c} S(t) = [v_1, \cdots, v_n] \qquad I = \{1, \cdots, n\} \\ S; f\{v_j/x\} \Downarrow S; \textbf{true} \qquad S; e\{v_j/x\} \Downarrow S; w_j \qquad j \in J \subseteq I \\ S; f\{v_k/x\} \Downarrow S; \textbf{false} \qquad k \in I \backslash J \\ S'(t) = [v_{k_1}, \cdots, v_{k_q}, w_{j_i}, \cdots, w_{j_p}] \end{array}}{S; \textbf{update}\ x\ \textbf{in}\ t\ \textbf{with}\ e\ \textbf{where}\ f \Downarrow S'; \textbf{true}} \quad \text{(Entity Update – 1)}$$

$$\frac{\begin{array}{c} S(t) = [v_1, \cdots, v_n] \qquad I = \{1, \cdots, n\} \\ S; f\{v_j/x\} \Downarrow S; \textbf{true} \qquad S; e\{v_j/x\} \Downarrow S; w_j \qquad j \in J \nsubseteq I \\ S; f\{v_k/x\} \Downarrow S; \textbf{false} \qquad \forall k.\ k \in I \\ S'(t) = [v_1, \cdots, v_2] \end{array}}{S; \textbf{update}\ x\ \textbf{in}\ t\ \textbf{with}\ e\ \textbf{where}\ f \Downarrow S'; \textbf{false}} \quad \text{(Entity Update – 2)}$$

For the sake of simplicity we present a rule for **from** expression with only one entity. For each value that expression e yields **true**, that value (w_j) is selected. The result value for the whole **select** expression is the set of values that the condition e yield **true**.

$$\frac{\begin{array}{c} S(t) = [v_1, \cdots, v_n] \qquad I = \{1, \cdots, n\} \\ S; e\{v_j/x\} \Downarrow S; \textbf{true} \qquad S; f\{v_j/x\} \Downarrow S; w_j \qquad j \in J \subseteq I \\ S; e\{v_k/x\} \Downarrow S; \textbf{false} \qquad k \in I \backslash J \end{array}}{S; \textbf{from}\ (\ x\ \textbf{in}\ t\ )\ \textbf{where}\ e\ \textbf{select}\ f \Downarrow S; [\overline{w}]} \quad \text{(Entity Select)}$$

The **count** expression rule evaluates an expression e to a list of values and returns the total number of values in that list.

$$\frac{S; e \Downarrow S'; [v_1, \cdots, v_n]}{S; \textbf{count}\ (e) \Downarrow S'; n} \quad \text{(Count Function)}$$

The **min** and **max** functions allow to calculate the minimum and maximum value of a list. Expression e evaluates to a list of values and then the resulting value corresponds to the minimum or maximum value in the list depending on the function applied.

$$\frac{S; e \Downarrow S'; \overline{v} \qquad \forall x \in \overline{v} : v \leq x \qquad v \in \overline{v}}{S; \textbf{min}\ (e) \Downarrow S'; v} \quad \text{(Minimum Function)}$$

$$\frac{S; e \Downarrow S'; \overline{v} \qquad \forall x \in \overline{v} : v \geq x \qquad v \in \overline{v}}{S; \textbf{max}\ (e) \Downarrow S'; v} \quad \text{(Maximum Function)}$$

Where $\overline{v} = [v_1, \cdots, v_n]$.

For a given sequence of blocks, each block b_i evaluates to a block value w_i and the

result is the set of values obtained, $\overline{w}$.

$$\frac{S_i; b_i \Downarrow S_{i+1}; w_i \qquad i = 1, \cdots, n}{S_1; \overline{b} \Downarrow S_{n+1}; \overline{w}} \quad \text{(Sequence of Blocks)}$$

The line break (**br**) rule is straight forward, since it does not contains any expressions or blocks that require evaluation.

$$\frac{}{S; \mathbf{br} \Downarrow S; \mathbf{br}} \quad \text{(Line Break)}$$

The **label** block evaluates an expression e to a value v resulting in a block value that shows the value obtained.

$$\frac{S; e \Downarrow S'; v}{S; \mathbf{label}\ e \Downarrow S'; \mathbf{label}\ v} \quad \text{(Label)}$$

The **div** block allows to define custom look to a block through a CSS class named $label$. The inner block b evaluates to a block value w and the result is the formated **div** containing w.

$$\frac{S; b \Downarrow S'; w}{S; \mathbf{div}\ label\ \{\ b\ \} \Downarrow S'; \mathbf{div}\ label\ \{\ w\ \}} \quad \text{(Div)}$$

The **image** block displays an image based on a given URL v evaluated from the expression e.

$$\frac{S; e \Downarrow S'; v}{S; \mathbf{image}\ e \Downarrow S'; \mathbf{image}\ v} \quad \text{(Image)}$$

The **link** block links a block to a given URL. The inner block b evaluates to a block value w and the expression e, which corresponds to the URL, evaluates to v. As result we have a **link** element linking the block value w to the URL value v.

$$\frac{S; b \Downarrow S'; w \qquad S'; e \Downarrow S''; v}{S; \mathbf{link}\ \{\ b\ \}\ \mathbf{to}\ e \Downarrow S''; \mathbf{link}\ \{\ w\ \}\ \mathbf{to}\ v} \quad \text{(Link)}$$

The **iterator** block is similar to the **foreach** expression. It evaluates an expression e to a list of values and iterates that list. For each value v_i in the list, the block b is evaluated to w_i with x replaced by v_i. The result is a set of all blocks values obtained.

$$\frac{S; e \Downarrow S_1; [v_1, \cdots, v_n] \qquad S_i; b\{v_i/x\} \Downarrow S_{i+1}; w_i \qquad i = 1, \cdots, n}{S; \mathbf{iterator}\ (x\ \mathbf{in}\ e)\ \{\ b\ \} \Downarrow S_{n+1}; \overline{w}} \quad \text{(Iterator)}$$

The **textfield** block binds user input to a local name x. This input can then be used to send data back to the server with the **button** block. Expression e evaluates to a value

v that corresponds to the initial value in the HTML input element.

$$\frac{S; e \Downarrow S'; v}{S; \textbf{textfield } x \textbf{ with } e \Downarrow S'; \textbf{textfield } x \textbf{ with } v} \quad \text{(Text Field)}$$

The **button** block creates an input button element with the label v_1 and when clicked executes the value associated to the expression e_2. This expression e_2 is used to call an action or screen in the server, sending input from text fields as argument to an action or screen, like a submit button in an HTML form. The execution of the web page interaction is not defined in this semantics so, the value associated to expression e_2 is simplified as much as possible (e.g. expression $1 + 2$ is simplified to 3) and converted to JavaScript to be included in the web page source. When the **button** is clicked, JavaScript code executes and solves open names from text fields and then issues a new request for the desired action or screen. Only JavaScript code is executed in the client, the code regarding the action or screen that is called is executed at the server.

$$\frac{S; e_1 \Downarrow S'; v_1}{S; \textbf{button } e_1 \textbf{ to } e_2 \Downarrow S'; \textbf{button } v_1 \textbf{ to } e_2} \quad \text{(Button)}$$

2.4 Type System

We now define the type system for our language by a rule system for the judgment with the standard form:

$$\Delta \vdash e : \mathcal{T}$$

Where $\mathcal{T}$ is a type, defined by the syntax in Figure 2.8, which is assigned to expression e with relation to the typing environment Δ. The type language comprises basic types, integers (Int), booleans ($Bool$), and strings ($String$). Composite types like lists ($List\langle\mathcal{T}\rangle$), structures ($\{\cdots\}$), and web page blocks ($Block$). List types describe homogeneous sequences of values, and structure types are labeled product types describing instances of entities (or rows of database tables).

The following rules describe the typing system for binary operators (logic, arithmetic, compare, and sequence). Logic operators require that both expressions have type $Bool$ and results in a type $Bool$

$$\gamma \in \{\textbf{and}, \textbf{or}\} \qquad \frac{\Delta \vdash e_1 : Bool \qquad \Delta \vdash e_2 : Bool}{\Delta \vdash e_1 \ \gamma \ e_2 : Bool} \quad \text{(Logic Operators)}$$

Arithmetic operators are only available to Int type expressions, so both expressions

must have type Int and also result in a type Int.

$$\phi \in \{+, -, *, /\} \qquad \frac{\Delta \vdash e_1 : Int \qquad \Delta \vdash e_2 : Int}{\Delta \vdash e_1 \; \phi \; e_2 : Int} \qquad \text{(Arithmetic Operators)}$$

Compare operators are divided in two groups. First group (θ) can be applied to any type of expressions and result in a $Bool$ type. The second group of operators (σ) require that both expressions have type Int and result in a type $Bool$.

$$\theta \in \{==, != \} \qquad \sigma \in \{>, >=, <, <=\}$$

$$\frac{\Delta \vdash e_1 : \mathcal{T} \qquad \Delta \vdash e_2 : \mathcal{T}}{\Delta \vdash e_1 \; \theta \; e_2 : Bool} \qquad \frac{\Delta \vdash e_1 : Int \qquad \Delta \vdash e_2 : Int}{\Delta \vdash e_1 \; \sigma \; e_2 : Bool} \qquad \text{(Compare Operators)}$$

The sequence operator (;) is similar to the binary operators above except the result type $\mathcal{T}_2$ corresponds to the type associated with the right-hand side expression e_2.

$$\frac{\Delta \vdash e_1 : \mathcal{T}_1 \qquad \Delta \vdash e_2 : \mathcal{T}_2}{\Delta \vdash e_1; e_2 : \mathcal{T}_2} \qquad \text{(Sequence Operator)}$$

Unary operator **not** can only be applied to boolean expressions and returns a $Bool$ type. The second unary operator ($-$) can only be applied to Int expressions and also returns a type Int.

$$\frac{\Delta \vdash e : Bool}{\Delta \vdash \textbf{not} \; e : Bool} \qquad \frac{\Delta \vdash e : Int}{\Delta \vdash -e : Int} \qquad \text{(Unary Operators)}$$

An identifier x has type $\mathcal{T}$ in a typing environment where x has type $\mathcal{T}$.

$$\frac{}{\Delta, x : \mathcal{T} \vdash x : \mathcal{T}} \qquad \text{(Identifier)}$$

Variable declaration allows to associate any value type to a label x. The inner expression e_2 has type $\mathcal{T}_2$ in a typing environment where x has type $\mathcal{T}_1$, obtained from expression e_1. The result type is the same as the type resulting from e_2.

$$\frac{\Delta \vdash e_1 : \mathcal{T}_1 \qquad \Delta, x : \mathcal{T}_1 \vdash e_2 : \mathcal{T}_2}{\Delta \vdash \textbf{let} \; x = e_1 \; \textbf{in} \; e_2 : \mathcal{T}_2} \qquad \text{(Variable Declaration)}$$

The conditional expression **if** takes an expression c, the condition, with type $Bool$ and both branches expressions, e_1 and e_2, must have the same type. The result type of the whole **if** expression is the same as the branches, $\mathcal{T}$.

$$\frac{\Delta \vdash c : Bool \qquad \Delta \vdash e_1 : \mathcal{T} \qquad \Delta \vdash e_2 : \mathcal{T}}{\Delta \vdash \textbf{if} \; c \; \textbf{then} \; e_1 \; \textbf{else} \; e_2 : \mathcal{T}} \qquad \text{(Condition)}$$

In the typing of action or screen call expressions, each argument expression e_i has type $\mathcal{T}_i$ that must match the parameter type in the corresponding definition. In the case of an action call, the result type $\mathcal{T}_r$ is obtained directly from the definition. Screens return type is always $Block$, therefore it is omitted in screen definitions.

$$\frac{P(a) = a(\overline{x : \mathcal{T}}) : \mathcal{T}_r \, \{ \cdots \} \qquad \Delta \vdash e_i : \mathcal{T}_i \qquad i = 1, \cdots, n}{\Delta \vdash a(\overline{e}) : \mathcal{T}_r} \quad \text{(Action Call)}$$

$$\frac{P(s) = s(\overline{x : \mathcal{T}}) \, \{ \cdots \} \qquad \Delta \vdash e_i : \mathcal{T}_i \qquad i = 1, \cdots, n}{\Delta \vdash s(\overline{e}) : Block} \quad \text{(Screen Call)}$$

Since supported lists are homogeneous sequence of values, all expressions e_i must have the same type $\mathcal{T}$, and as a final result the list constructor is a list of values with type $\mathcal{T}$.

$$\frac{\Delta \vdash e_i : \mathcal{T} \qquad i = 1, \cdots, n}{\Delta \vdash [\, \overline{e} \,] : List\langle \mathcal{T} \rangle} \quad \text{(List)}$$

The **foreach** iterator takes an expression e_1 with type $List\langle \mathcal{T}_1 \rangle$ and the inner expression e_2 has type $\mathcal{T}_2$ in a typing environment where x has type $\mathcal{T}_1$ (the inner type of the list obtained from e_1).

$$\frac{\Delta \vdash e_1 : List\langle \mathcal{T}_1 \rangle \qquad \Delta, x : \mathcal{T}_1 \vdash e_2 : \mathcal{T}_2}{\Delta \vdash \textbf{foreach } x \textbf{ in } e_1 \textbf{ do } e_2 : List\langle \mathcal{T}_2 \rangle} \quad \text{(List Iterator)}$$

A structure is built based in a set of expressions e_i, where each expression e_i is associated with label l_i. Each expression e_i has type $\mathcal{T}_i$ and two distinct expressions may have different types.

$$\frac{\Delta \vdash e_i : \mathcal{T}_i \qquad i = 1, \cdots, n}{\Delta \vdash \{\, \overline{l = e} \,\} : \{\, \overline{l : \mathcal{T}} \,\}} \quad \text{(Structure)}$$

To access a structure field, the expression e must be a structure and contain the required field named $label$. The result type is the same as the type associated to $label$.

$$\frac{\Delta \vdash e : \{\cdots, label : \mathcal{T}, \cdots\}}{\Delta \;\vdash\; e.label : \mathcal{T}} \quad \text{(Structure Field)}$$

The typing of the **insert** expression requires that t is an entity with rows of type $\mathcal{T}$ (represented as $List\langle \mathcal{T} \rangle$) and the expression e must evaluate to the same type $\mathcal{T}$. The result type is boolean stating the successfulness of the **insert** operation.

$$\frac{\Delta \vdash t : List\langle \mathcal{T} \rangle \qquad \Delta \vdash e : \mathcal{T}}{\Delta \vdash \textbf{insert } e \textbf{ in } t : Bool} \quad \text{(Entity Insert)}$$

The **update** expression for an entity t with rows of type $\mathcal{T}$ (represented as $List\langle\mathcal{T}\rangle$) performs an update based on an expression e of type $\mathcal{T}$ with a boolean conditional expression f. The result is also $Bool$ stating if any record from the entity t was modified.

$$\frac{\Delta \vdash t : List\langle\mathcal{T}\rangle \qquad \Delta, x : \mathcal{T} \vdash e : \mathcal{T} \qquad \Delta, x : \mathcal{T} \vdash f : Bool}{\Delta \vdash \textbf{update}\ x\ \textbf{in}\ t\ \textbf{with}\ e\ \textbf{where}\ f : Bool} \qquad \text{(Entity Update)}$$

The **from** expression rule is simplified for a single entity, and it selects a set of records from one entity t. The selected records are filtered according to a conditional expression e, i.e., for each record that the conditional expression e is **true** that record is selected. The result expression f has type U and hence the result of the **from** expression is $List\langle U\rangle$.

$$\frac{\Delta \vdash t : List\langle\mathcal{T}\rangle \qquad \Delta, x : \mathcal{T} \vdash e : Bool \qquad \Delta, x : \mathcal{T} \vdash f : U}{\Delta \vdash \textbf{from}\ (x\ \textbf{in}\ t)\ \textbf{where}\ e\ \textbf{select}\ f : List\langle U\rangle} \qquad \text{(Entity Select)}$$

As for the **count** expression, it results in an Int type and is available for any list.

$$\frac{\Delta \vdash e : List\langle\mathcal{T}\rangle}{\Delta \vdash \textbf{count}\ (e) : Int} \qquad \text{(Count Function)}$$

The **min** and **max** expressions are only available for lists of integers and both return an Int type.

$$\frac{\Delta \vdash e : List\langle Int\rangle}{\Delta \vdash \textbf{min}\ (e) : Int} \qquad \frac{\Delta \vdash e : List\langle Int\rangle}{\Delta \vdash \textbf{max}\ (e) : Int} \qquad \text{(Minimum and Maximum Functions)}$$

The type system rules for blocks always result in a type $Block$, an interface element. We present the block rules in a head and tail form, where the head is the first block being evaluated and the tail is the remaining blocks (possibly none). With this kind of presentation we omit the rule for sequence of blocks, since the sequence $\overline{b}$ is represented in every rule and has always type $Block$. In the following rules for blocks we explain the block being evaluated, since the tail is always $\overline{b}$ with type $Block$.

The **br** block typing rule is straight forward since it does not contains any inner expressions or blocks.

$$\frac{\Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{br}, \overline{b} : Block} \qquad \text{(Line Break)}$$

For the **label** block the expression to be printed in the web page e can be of any type, hence $\mathcal{T}$.

$$\frac{\Delta \vdash e : \mathcal{T} \qquad \Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{label}\ e, \overline{b} : Block} \qquad \text{(Label)}$$

The **div** block has an inner block c that must have type $Block$. The formatting CSS class $label$ is defined as a string by the language syntax.

$$\frac{\Delta \vdash c : Block \qquad \Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{div}\ label\ \{\ c\ \}, \overline{b} : Block} \tag{Div}$$

The typing rule for the **image** block takes an expression e that corresponds to the image URL, this URL can be of any type.

$$\frac{\Delta \vdash e : \mathcal{T} \qquad \Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{image}\ e, \overline{b} : Block} \tag{Image}$$

The **iterator** typing rule takes an expression e of type $List\langle\mathcal{T}\rangle$ and the inner block c has type $Block$ in a typing environment where x has type $\mathcal{T}$.

$$\frac{\Delta \vdash e : List\langle\mathcal{T}\rangle \qquad \Delta, x : \mathcal{T} \vdash c : Block \qquad \Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{iterator}\ (x\ \textbf{in}\ e)\ \{\ c\ \}, \overline{b} : Block} \tag{Iterator}$$

The **textfield** block takes an expression e of any type $\mathcal{T}$ and then the tail sequence $\overline{b}$ is evaluated in a typing environment where x has type $\mathcal{T}$, the same type as the expression e.

$$\frac{\Delta \vdash e : \mathcal{T} \qquad \Delta, x : \mathcal{T} \vdash \overline{b} : Block}{\Delta \vdash \textbf{textfield}\ x\ \textbf{with}\ e, \overline{b} : Block} \tag{Text Field}$$

The expressions in **link** and **button** elements that represent context switching in the browser (after the token **to**) must always yield web page block (action or screen call) or an expression that has type $String$ (external URL).

$$\frac{\Delta \vdash c : Block \qquad \Delta \vdash e : \mathcal{T} \qquad \Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{link}\ \{\ c\ \}\ \textbf{to}\ e, \overline{b} : Block} \tag{Link}$$

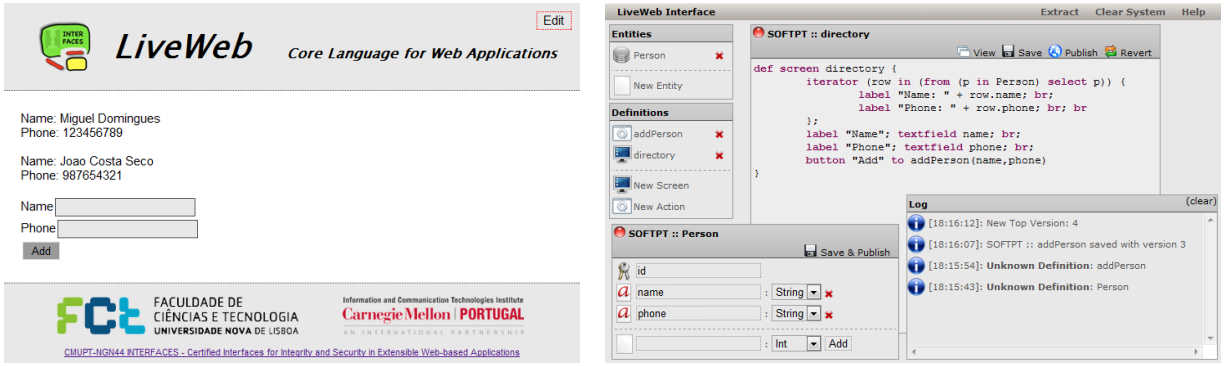
$$\frac{\Delta \vdash e_1 : \mathcal{T}_1 \qquad \Delta \vdash e_2 : \mathcal{T}_2 \qquad \Delta \vdash \overline{b} : Block}{\Delta \vdash \textbf{button}\ e_1\ \textbf{to}\ e_2, \overline{b} : Block} \tag{Button}$$

3

Runtime Support System

In addition to design a core language for web applications, this work also aims at implementing a prototype for our language. This prototype aims at being an interpreter for the language and at the same time a development environment. We consider the two interaction modes depicted in Figure 3.1, the *execution mode* and the *development mode*. Given the correct URL in the execution mode the user sees the screens defined in the application mode, just like a common web application. In development mode the user sees a code editor and an application browser, although all operations are made through a web based environment.

Our prototype architecture is depicted in Figure 3.2 and includes an HTTP server, a language interpreter, a type checker, and two databases, one for the application data (data created and managed by the application itself) and another for the application code (contains all saved versions of application elements). The system entry point is the HTTP server, which receives requests from the browser and either manages and checks the code (development mode) or orders the language interpreter to execute some action or screen (execution mode).



(a) Execution Mode (b) Development Mode

Figure 3.1: Prototype Interaction Modes

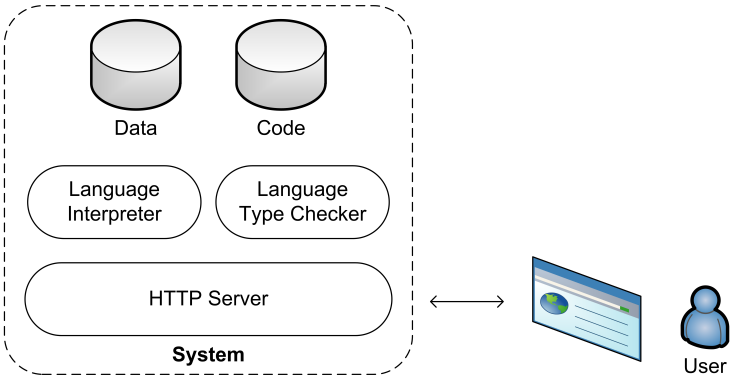


Figure 3.2: System Architecture

3.1 Execution Mode

Applications running in the system are accessed through well determined URLs[1] containing the information about the module name (that works like a namespace), a programming element name (action or screen) and the corresponding arguments. By using a different module name for each application we can have several applications running in the same system, without any kind of interference between applications.

An action or screen can be parameterized and the argument values are encoded and sent though the URL. Since a URL is a string, when the HTTP server receives a request, it must get all parameters and check if the requested action or screen parameter types match. If a parameter type is not a *String*, then the argument value must be converted to the expected type, but depending on the value sent to the server, it might not be possible to perform the cast operation. If this happens, then the execution is aborted and nothing is modified in the server. Upon successful cast of all values, the action or screen execution can proceed. The same happens when an action or screen is called

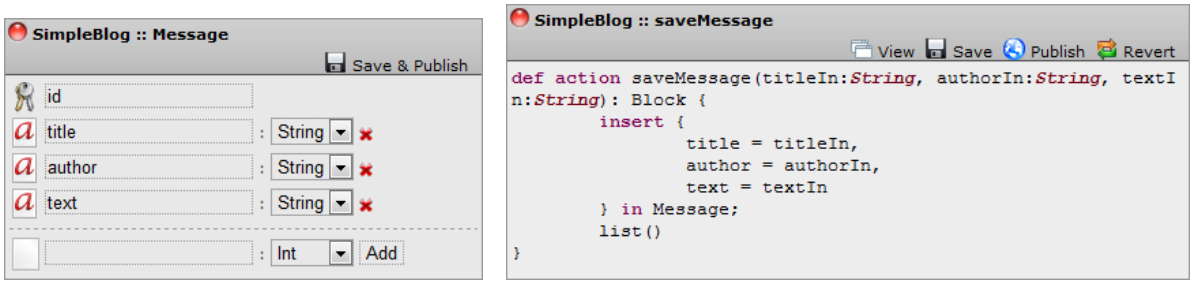
[1] http://server:port/module/element/arg0/arg1/.../

through **button** or **link** elements: all the data received on the server end are strings and if needed cast operations are performed. Notice that generated code produces well typed calls in web pages. These type conversions are only needed in hand crafted URLs.

When a valid HTTP request is issued, the system parses the URL gathering information about the module name, element name, and its arguments. Based on the gathered information, the system loads the application from the specified module and executes the desired element (screen or action) with the arguments received (if any). The code of the running application is determined by the latest published version of each programming element. The system ensures that the latest published version of all elements from the same application is sound. Screens are all rendered using common application layout, hard wired in the web server (Figure 3.1a). This layout contains in the top right corner an *Edit* button that switches from the execution mode to the development mode. Switching to development mode allows to modify the definition of the current screen. We designed this prototype as an open development environment and we have not encoded any protection mechanisms that must be obviously implemented in normal operation mode.

Each screen consists in one web page, which is rendered in standard HTML and JavaScript. When the interpreter reaches a web page block (e.g., **br**, **div**, **image**) it evaluates all inner expressions to values and then the block being evaluated is rendered as pure HTML code. JavaScript code is used to build the expressions that go back to the server, i.e., screen or action calls in expressions after the token **to** in **link** and **button** blocks. If the screen or action call contains arguments, those expressions must be evaluated to values, only open names referring text fields are left unsolved and is here that JavaScript is inserted. Before calling the action or screen, values from **textfield** elements are obtained using JavaScript and included in the request. Although challenging the current implementation is quite limited and only allows action and screen calls to go back to the server. A richer language can be used in these interaction spots by means of AJAX technology but this lies out of the scope of this work.

The language interpreter upon reaching a query expression converts it to SQL. During this conversion all inner expressions are evaluated to values and those values converted to database types and included in the query and, only then, the query is executed in the database. For an **update** expression the result depends on the number of updated rows, and for an **insert** expression the result depends on the successfulness of the insertion. In the case of a **from** expression the query results are read and each row is converted to a structure, and all structures joined in one list. This list is now a value that belongs to our language and can be manipulated by the language expressions.



(a) Entity Editor (b) Code Editor

Figure 3.3: Editing Windows

3.2 Development Mode

Our web based development environment lets the user access and change the elements of an application, with an operation close to a developer Wiki system. The environment (Figure 3.1b) lists all available entities, screens, and actions for a given module (which is identified by the first fragment of the URL). The environment then opens the code of each element inside the browser window and shows a console in the lower right corner. Figure 3.3a displays the editing window for an entity and Figure 3.3b displays the editing window for an action (screen editing window is similar). The system also makes some other commands available in the top right corner, like the *Extract* command, which allows to view the complete definition of the active application, the *Clear System* command, which allows anyone to delete all definitions and reset the system, and the *Help* command that shows the language specification.

The runtime support system has the notion of saved and published version of application definitions. Every time a definition is saved, a version number is associated to it. Upon publishing, if the whole application is considered sound, by the type checker, then that version is accepted and considered active or published. This published version corresponds to the application definition that is actually running in execution mode.

Existing entities can be modified by adding new attributes, change existing attributes, or delete existing attributes. These operations are available through a specific editing window for entities (Figure 3.3a). This window also allows to *Save & Publish* the new entity definition. We opted not to allow only to save entities and force publish, since entities are one of the most important components of web applications, and most modifications in entities force modifications in other components (actions and screens).

Actions and screens are defined using a window similar to a text editor (Figure 3.3b) where language syntax tokens are highlighted. Each window has several commands:

the *View* command allows to view the current action or screen published version in execution mode, *Save* and *Publish* buttons allow to perform each operation independently, and the *Revert* button allows to revert the current definition to the last published version code (if it exists).

The dynamic reconfiguration mechanism implemented ensures that after publishing a new definition, the system loads the new application definition and performs the necessary modifications ensuring that the application runs as expected. When a screen or action is modified or created the only reconfiguration needed is to load the new screen or action definition. Although when an entity is modified that is not the case, the *Data* database must also be modified to match the new entity definition. If a new attribute is added to an entity, then the database table is modified to include a new column corresponding to that attribute. The opposite operation, removing an attribute, is similar, the corresponding column is removed from the database table. Changing the type of an entity attribute causes a type cast on all existing values to the new type. In either cases, the existing data in entities is kept, ensuring that the application evolves to a new state where the previous existing data is still present. When a new entity is defined, the *Data* database is also updated, a new table corresponding to the new entity is created.

4

Web Applications Development

The main-stream of web applications development is based on a three layer architecture that divides applications into client interface, business logic, and database layers. In practice, applications are developed in heterogeneous programming language environments, and in particular, business logic is specified using general purpose programming languages to define computations and specialized query languages to access information in databases [16].

General purpose programming languages can be divided into compiled languages (e.g., Java, C#) and interpreted languages (e.g., PHP, Ruby, ASP) [23]. In compiled languages the application code is "translated" (compiled) to machine code for direct execution by the hardware or, in some cases the code is translated to an intermediate language (e.g., Java Bytecode), and then executed using a virtual machine (e.g., Java Virtual Machine). Despite offering static verifications and optimizations in compile time, when the application code is modified it is required to stop the application execution, recompile the new code and then start the application. Interpreted languages or scripting languages do not have a compiler or compilation phase, instead these programming languages rely on an interpreter that executes instructions directly from the application code.

It is also important to refer another difference between compiled and interpreted languages, specially when used in web applications environments: dynamic typing

versus static typing. In a dynamically typed language (e.g., Ruby, PHP, ASP), variable types are detected during execution according to their value, causing type errors to be detected only when the application executes the code, i.e., when a browser request is issued. Static typed languages (e.g., Scala, Java) have types associated with variables (as opposed to values) or the language supports type inference (types are discovered by the compiler based on variables context).

Usually compiled languages are associated with static typing, and interpreted languages associated with dynamic typing or no typing at all. In compiled languages, during the compile process, static type checks ensure that the application behavior will be correct during execution, avoiding constant repetition of type checks during execution, as happens with interpreted languages.

Even with a compiled language, which offers greater security over the correct application behavior, knowledge of other technologies is also required in order to be able to integrate the three layers of a common web architecture. Interaction with the database usually requires SQL knowledge. On the other side, an interface may require HTML and JavaScript knowledge. Therefore, to create a complete web application with a three layer architecture, it would be required to use several languages that do not have any real connection between them, and without verifications in the communication code between layers.

Ruby On Rails (Section 4.1), CakePHP (Section 4.2), Scala Lift (Section 4.3), and Google Web Toolkit (Section 4.4) frameworks target at web applications providing scaffolding features to increase developers productivity, while others (Hibernate, LINQ, and ScalaQL) provide extensions to general purpose languages and include typing for database operations (Section 4.5). Links (Section 4.6), WebDSL (Section 4.7), Ur/Web (Section 4.8), and Agile Platform (Section 4.9) belong to a third category of frameworks that use domain specific languages to provide program safety by construction.

4.1 Ruby On Rails

Ruby On Rails framework is based on the Ruby[1] programming language, which is an interpreted language with support for several paradigms (functional, object-oriented, and imperative) and is dynamically typed [20]. Rails framework architecture is based on the Model-View-Controller (MVC) pattern [27], where *ActiveRecord*, *ActionView*, and *ActionController* are the three components from the MVC pattern.

Figure 4.1 illustrates how an HTTP request is processed within the Rails framework:

1. The request is made by the user's browser;

[1] http://www.ruby-lang.org/

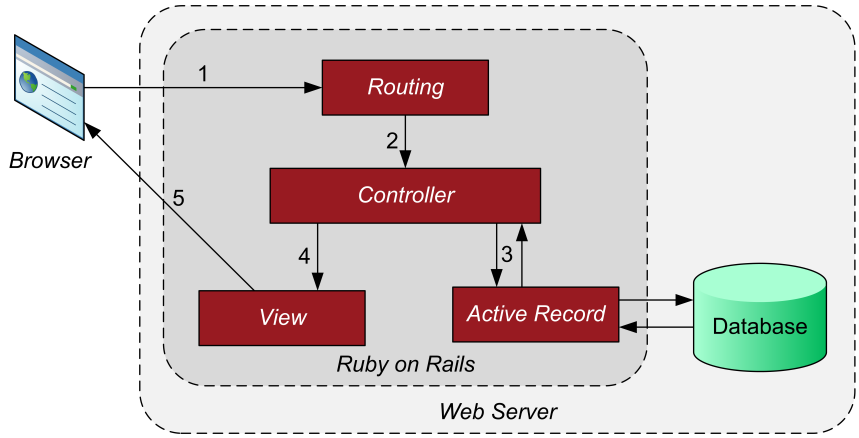


Figure 4.1: Ruby On Rails Request Processing [29]

```
class Message < ActiveRecord::Base
end
```

Listing 4.1: Message Model

2. *Routing* component transforms the URL in a request to a *Controller*;

3. Interaction with the *Model* (database) through the *ActiveRecord* pattern;

4. Call to the *View*;

5. Interface is rendered and the result sent to the browser.

The *ActiveRecord* component is an Object-Relational Mapping (ORM) abstraction that controls the read and write operations in the database, turning the developed code independent from the Database Management System (DBMS). This approach allows portability between several supported DBMS, like MySQL, PostgreSQL, Oracle, DB2, or SQLite [23,29] without modifying the application code. ORM abstraction turns each database table row into a Ruby object, e.g., a table named Message with five rows will match five different objects of type Message (Listing 4.1) in the application. The abstraction used also supports table relationships such as `1:1`, `1:n` and `n:n` [29], input data validation, and a version system that keeps track of the modifications made in the database model (Listing 4.2), allowing to rollback to a previous version or update to a higher version [23]. To create a new database table, the developer simply creates an *ActiveRecord* and a migration for that table, the ORM abstraction then performs the required modifications in the database applying the specified database migration.

The bridge between the model (*ActiveRecord*) and the interface (*ActionView*), i.e., the business logic is defined in the controller (Listing 4.3), which collects data sent by the

```
class CreateMessages < ActiveRecord::Migration
  def self.up
    create_table :messages do |t|
      t.string :title
      t.string :author
      t.date :date
      t.text :text

      t.timestamps
    end
  end

  def self.down
    drop_table :messages
  end
end
```

Listing 4.2: Database Migration

browser and sets how that data is used in the request, it also reads and writes data from the database through the ORM abstraction and sends the data to the *ActionView*.

The interface code that defines what is sent to the browser is defined in *ActionViews* (Listing 4.4, 4.5 and 4.6), and can be plain HTML or a mix of HTML and Ruby (embedded Ruby) [23].

Besides the three components from the MVC pattern, there is also a set of *Helpers* that consists in reusable pieces of code. Rails framework provides a default set of *Helpers* that allows developers to create forms, manipulate URLs, and date/time values [23]. There is also a scaffolding generator that creates every MVC component including a CRUD[2] interface for a given data model [23, 29]. The framework provides different environments for application development, test, and production, allowing to isolate each task:

- **Development.** Modifications in the application code are immediately visible in the browser and some detailed logs are produced with details about the application execution (useful when new features are being developed).

- **Test.** In this environment the database is filled with dummy data, and tests defined by the developer, ensuring that the results of the tests are consistent, and the application behavior is reproducible. It is possible to test models (*Unit Tests*), actions in controllers (*Functional Tests*), and test the flow between controllers (*Integration Tests*) [23, 29].

- **Production.** Environment where changes in the application are less frequent and

[2]Create, Read, Update, Delete

```
class MessagesController < ApplicationController
  def index
    if (params[:author])
      @messages = Message.all(:conditions => { :author => params[:author]})
    else
      @messages = Message.all
    end
  end

  def show
    @message = Message.find(params[:id])
  end

  def new
    @message = Message.new
  end

  def create
    @message = Message.new(params[:message])
    if @message.save
      flash[:notice] = 'Message was successfully created.'
      redirect_to(@message)
    else
      render :action => "new"
    end
  end
end
```

Listing 4.3: Blog Controller

```
<h1>Message List</h1>
<table border="1">
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>View</th>
  </tr>
<% @messages.each do |message| %>
  <tr>
    <td><%=h message.title %></td>
    <td><%=h message.author %></td>
    <td><%= link_to 'View', message %></td>
  </tr>
<% end %>
</table>
<%= link_to 'Write Message', new_message_path %>
```

Listing 4.4: "Message List" Interface

for performance reasons tasks like logging are disabled.

On each of the environments a different database is used, avoiding changes made in the development environment to be visible in the production environment [23]. It

```
<h1>New message</h1>
<% form_for(@message) do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :author %><br />
    <%= f.text_field :author %>
  </p>
  <p>
    <%= f.label :date %><br />
    <%= f.date_select :date %>
  </p>
  <p>
    <%= f.label :text %><br />
    <%= f.text_area :text %>
  </p>
  <p>
    <%= f.submit 'Create' %>
  </p>
<% end %>
```

Listing 4.5: "Add Message" Interface

```
<h1><%=h @message.title %></h1>
<h2> by <%=h @message.author %> at <%=h @message.date %></h2>
<p><%=h @message.text %></p>
More messages from <%= link_to @message.author, {
  :action => "index", :author => @message.author }
%>
```

Listing 4.6: "View Message" Interface

is also possible to use different DBMS in each environment, the *ActiveRecord* (ORM) abstraction makes sure that the application works with every DBMS in the same way. Rails also supports features such as session data and cookies management, AJAX, and REST[3].

The example presented in this section is similar to the example described in Section 2.1.2 for our core language. This simple blog example allows any user to post text messages. Listing 4.1 and 4.2 display the definitions for the database layer. Comparing to the approach in our language, Rails supports database migrations, where we only can modify entities and publish them, not allowing rollback operations. However, both Rails and our language perform an automatic mapping of entities (models) definitions to the database. Querying an entity in our language uses a syntax similar

[3]Representational State Transfer

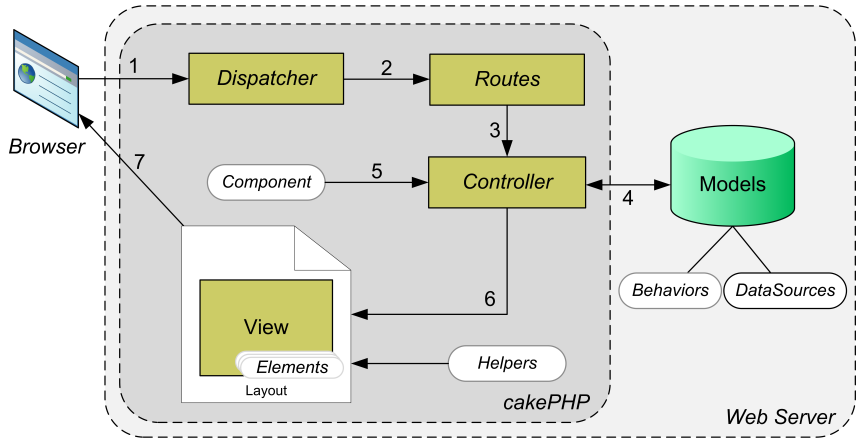


Figure 4.2: CakePHP Request [1]

to SQL, while Rails uses an ORM abstraction with its own querying language. In Rails, controllers (Listing 4.3) contain the equivalent to a set of actions from our language. The controller defines the application business logic, but each definition (index, show, and new) is associated to a user interface with the same name or the definition (create) specifies which user interface is rendered. The interface definition in Rails is a mix between HTML and Ruby (Listing 4.4, 4.5, and 4.6) while our language has support to define interface elements using built-in blocks. The major difference between Rails and our language is static check support. While Rails has no support for static checks and errors are only detected at runtime, our language supports static checks avoiding common programming errors in unverified code, in particular, in communication code between layers.

4.2 CakePHP

CakePHP framework [11] is inspired in Ruby On Rails and shares some of the key features like the Model-View-Controller pattern [1, 27]. Applications developed using this framework, and the framework itself, use the PHP[4] programming language. This scripting language joins the imperative and the object-oriented paradigm and is dynamically typed.

Figure 4.2 demonstrates how a request is processed using the CakePHP framework. The process starting from the request until the reply is as follows:

1. HTTP request to the server;

2. *Controller*, action and parameters are extracted from the requested URL;

[4] http://www.php.net/

```
CREATE TABLE `cake`.`messages` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(255) NOT NULL,
  `author` VARCHAR(255) NOT NULL,
  `date` DATE NOT NULL,
  `text` TEXT NOT NULL,
PRIMARY KEY (`id`)
) ENGINE = MYISAM;
```

Listing 4.7: MySQL Table Creation

```
<?php
class Message extends AppModel {
  var $name = 'Message';
}
?>
```

Listing 4.8: Blog Model

3. The desired controller is called through defined routes;

4. Data exchange with *Models* (e.g., database);

5. *Components* are used to execute common tasks to several controllers;

6. When all data is processed by the *Controller*, the web page layout (*View*) is built using *Helpers* if needed;

7. The final result is sent to the browser.

Similar to Rails, the Model-View-Controller pattern is represented by the *Model* (Listing 4.8), *View* (Listing 4.10, 4.11, and 4.12), and *Controller* (Listing 4.9) components in CakePHP framework. The major difference between Rails and CakePHP is that, in CakePHP, database tables must be manually created using SQL and a *Model* for each table (Listing 4.7 and 4.8) must also be created manually. Still related to *Models*, they do not support multi-column keys but support `1:1`, `1:n`, and `n:n` table relationships and validation rules for each column in the tables. It is also possible to use several Database Management Systems such as MySQL, PostgreSQL, SQLServer, or ODBC [6].

MVC components can also be extended, *Models* are extended through *Behaviors* allowing to add common features to one or more *Models*, and *DataSources* are abstractions to manipulate different sources of data (e.g., databases, RSS, CSV, LDAP, iCal). *Views* are extended through *Helpers*, i.e., reusable code for *Views*. *Controllers* can also be extended using *Components*, i.e., common business logic for controllers or applications [1].

```
<?php
class MessagesController extends AppController {
  var $name = 'Messages';
  function index() {
    $this->set('messages', $this->Message->find('all'));
  }
  function view($id) {
    $this->Message->id = $id;
    $this->set('message', $this->Message->read());
  }
  function add() { }
  function save(){
    if ($this->Message->save($this->data)) {
      $this->Session->setFlash('Message saved.');
      $this->redirect(array('action' => 'view', $this->Message->id));
    }
  }
  function filter($author){
    $this->set('messages', $this->Message->findAllByAuthor($author));
    $this->render('index');
  }
}
?>
```

Listing 4.9: Blog Controller

```
<h1>Add Message</h1>
<?php
echo $form->create('Message', array("action" => "save"));
echo $form->input('title');
echo $form->input('author');
echo $form->input('date');
echo $form->input('text', array('rows' => '3'));
echo $form->end('Submit');
?>
```

Listing 4.10: "Add Message" Interface

CakePHP also has features like scaffolding (create CRUD interfaces), REST support and, session and cookies manipulation. Related to the interface, CakePHP offers *Helpers* for AJAX and JavaScript, date/time manipulation and forms [1,6].

The example presented throughout this section defines a simple blog where any user can post messages, similar to the example described in Section 2.1.2 for our language. In CakePHP, database tables are created manually (Listing 4.7) and then a model for each table must also be created (Listing 4.8). The approach in our language is simpler, since the developer only needs to define the entities and the database tables are created by the dynamic reconfiguration mechanism. As for controllers (Listing 4.9), CakePHP approach and Rails approach are similar, both have controllers (a set

```
<h1>Message List</h1>
<table>
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>View</th>
  </tr>
  <?php foreach ($messages as $message): ?>
  <tr>
    <td><?php echo $message['Message']['title']; ?></td>
    <td><?php echo $message['Message']['author']; ?></td>
    <td>
      <?php echo $html->link("View",
      array(
        'controller' => 'messages',
        'action' => 'view', $message['Message']['id']));
      ?>
    </td>
  </tr>
  <?php endforeach; ?>
</table>
<?php
echo $html->link("Write Message",
  array('controller' => 'messages', 'action' => 'add'));
?>
```

Listing 4.11: "Message List" Interface

```
<h1><?php echo $message['Message']['title']?></h1>

<h2>by <?php echo $message['Message']['author']?> at
  <?php echo $message['Message']['date']?></h2>

<?php echo $message['Message']['text']?>

<br />
More messages from <?php echo $html->link($message['Message']['author'],
  array('controller' => 'messages',
  'action' => 'filter', $message['Message']['author']));
?>
```

Listing 4.12: "View Message" Interface

of actions in our language) and each controller function has a corresponding interface screen (Listing 4.10, 4.11, and 4.12). In our language actions are not associated to any specific screen, if an action needs to render a screen just calls the screen in the action body. Since CakePHP uses an interpreted language it does not support static checks, while our language has support for them.

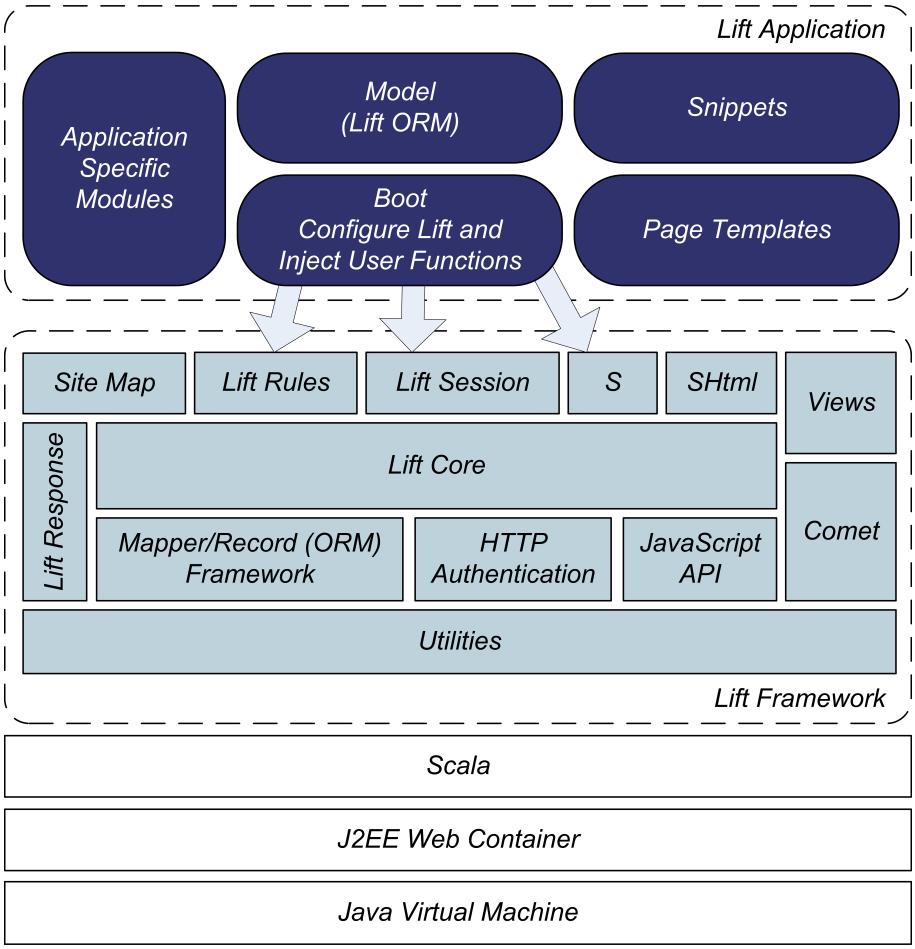


Figure 4.3: Scala Lift Architecture [12]

4.3 Scala Lift

Scala Lift framework [12] for web applications is based in the Scala[5] programming language. The code developed in this language is compiled to Java Bytecode and then executed in a Java Virtual Machine. In addition to be able to use libraries from Java, Scala joins the functional paradigm to the imperative and object-oriented paradigms from Java. Scala also supports local type inference, native XML processing, and functions as objects. Usually web applications developed with Lift execute in a Web component like Jetty[6] or Tomcat[7] [12].

Lift architecture is quite complex (Figure 4.3), however it is also based in the Model-View-Controller pattern [27], offering a clear separation between the interface and business logic, unlike Ruby On Rails, CakePHP, and JSP[8] that mixes business logic

[5] http://www.scala-lang.org/
[6] http://jetty.codehaus.org/jetty/
[7] http://tomcat.apache.org/
[8] Java Server Pages

code with the interface definition [12]. We now explain each component of the architecture:

- **LiftCore.** Framework engine responsible for receiving HTTP requests and creating responses.
- **SiteMap.** Application web pages. Among other things, allows to define access to menu items, group existing pages similar to a tree, and create web site navigation menus.
- **LiftRules.** Object to store application configuration settings.
- **LiftSession.** User session representation. Allows to manage information about each user session.
- **S.** Object that represents the state of the current request. Allows to manage cookies and the timezone, change and access HTTP headers from the request, set response HTTP headers, etc.
- **SHtml.** Helper functions to create forms and to use AJAX and Comet.
- **Views.** *LiftView* objects that contain the interface definition (XML).
- **LiftResponse.** Abstraction of a response that will be sent to the user (HTTP codes: 200 OK, 404 Not Found, etc.).
- **Comet.** Layer that sends asynchronous content to the browser.
- **Mapper/Record ORM.** Mapping between Scala objects and the relational model.
- **HTTP Authentication.** User authentication using Basic or Digest HTTP.
- **JavaScript API.** JavaScript abstraction layer to build JavaScript code within the application.
- **Utilities.** Helper functions used by the application and available to developers.

The *Model*, *Page Template*, and *Snippet* application components correspond to the Model, View, and Controller from MVC pattern [12, 27]. The *Boot* component is executed once when the application starts and contains the required configuration to start the application. This component defines URL parsing rules, database connections (default uses JNDI[9]), site map for navigation menus, etc. [12]. During application start,

[9] Java Naming and Directory

```
class Message extends LongKeyedMapper[Message] with IdPK {
  def getSingleton = Message

  object title extends MappedString(this,255)
  object author extends MappedString(this,255)
  object date extends MappedDateTime(this)
  object text extends MappedText(this)
}

object Message extends Message with LongKeyedMetaMapper[Message]
```

Listing 4.13: Blog Model

```
<h1>Message List</h1>
<table border="1">
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>View</th>
  </tr>
  <lift:Messages.list>
    <tr>
      <td><message:title /></td>
      <td><message:author /></td>
      <td><message:view /></td>
    </tr>
  </lift:Messages.list>
</table>
<a href="/blog/add">Write Message</a>
```

Listing 4.14: "Message List" Interface

the *Schemifier* mechanism inspects the defined *Models* and the current database configuration, and ensures that tables, columns, indexes, and integrity restrictions match between the two models. *Schemifier* mechanism can execute in manual mode where modifications are suggested, or in automatic mode where the mechanism performs the required modifications [12].

The *Models* are defined through inheritance from classes such as LongKeyedMapper, that map the defined models to the relational database (Object-Relational Mapping – ORM). Each object of type Message (Listing 4.13) maps to a row from the database table Message [12]. Columns are also defined by inheritance and can contain behavior like default values and validation rules. Inheritance can also be used to create more specific column types, e.g., define a type Email instead of using type String for a column email (in the type Email the value is converted to lower case) [12].

Lift framework has an approach where the interface definition should not contain any business logic code, however the interface definition (Listing 4.14, 4.15, 4.16, and

```
<lift:Messages.view>
  <h1><message:title /></h1>
  <h2> by <message:author /> at <message:date /></h2>
  <message:text /><br />
  More messages by <message:authorlink />
</lift:Messages.view>
```

Listing 4.15: "View Message" Interface

```
<h1>Message List</h1>
<table border="1">
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>View</th>
  </tr>
  <lift:Messages.filter>
    <tr>
      <td><message:title /></td>
      <td><message:author /></td>
      <td><message:view /></td>
    </tr>
  </lift:Messages.filter>
</table>
<a href="/blog/add">Write Message</a>
```

Listing 4.16: "Filter Message" Interface

4.17) must be flexible enough to support dynamic content. The framework combines the MVC pattern with the *View-First* pattern, i.e., each request matches a *View* or template (defined in XML) and specifies which controllers are executed to build the reply. This *View-First* approach allows to have a higher modularity of the business logic components. Usage of XML templates is only possible due the native XML processing in Scala and allows to embedded templates and insert dynamic content in each request. XML templates use specific tags (<lift:xyz />) to execute business logic code. Each tag matches a *Snipped*–method pair, i.e., a business logic method [12].

Scala is a compiled language, so a compilation step is required, although the XML templates are not statically checked during compilation, so errors in XML templates are not detected until execution time.

Snippets (Listing 4.18) correspond to the controllers in the MVC pattern and consist in a set of methods that return a NodeSeq object (sequence of XML/HTML nodes). Methods from *Snippets* are called when a XML template is executed and contains a tag similar to <lift:Snippet.method />. By returning a NodeSeq object, the method return value replaces the tag in the template, i.e., the method returns the XML/HTML code to be included in the reply sent to the browser [12].

```
<lift:Messages.add form="post">
  <table>
    <tr>
      <td>Title:</td>
      <td><message:title /></td>
    </tr>
    <tr>
      <td>Author:</td>
      <td><message:author /></td>
    </tr>
    <tr>
      <td>Date:</td>
      <td><message:date /></td>
    </tr>
    <tr>
      <td>Text:</td>
      <td><message:text /></td>
    </tr>
    <tr>
      <td rowspan="2"><message:submit /></td>
    </tr>
  </table>
</lift:Messages.add>
```

Listing 4.17: "Add Message" Interface

The example presented throughout this section is also similar to the example defined in our language from Section 2.1.2 where any user can post messages. Lift framework also uses an ORM abstraction like Rails and CakePHP. Although models are defined in a similar way, in comparison, to our language. This framework uses a compiled language with static check support like our language. Although views (Listing 4.14, 4.15, 4.16, and 4.17) are defined in XML and not subject to any static verifications in compile time. Business logic is defined in controllers (Listing 4.18) and methods inside controllers may contain XML code, mixing interface and business logic layers. However, XML code inside controllers is subject to static checks.

4.4 Google Web Toolkit

A greater browsing experience has been one of the most important aspects in web development. Users browsing experience gets higher with highly interactive applications. This kind of web applications are mainly developed using Asynchronous JavaScript and XML (AJAX), however differences among the several browsers has been a problem for developers. Each browser implements in it is own way JavaScript instructions such as XMLHttpRequest, highly used in AJAX applications.

```
class Messages {
  def list(html:NodeSeq): NodeSeq = {
    Message.findAll.flatMap(message =>
      bind("message", html,
                "title" -> Text(message.title.is),
                "author" -> Text(message.author.is),
                "view" -> <a href={"/blog/view/" +
                  message.id.is}>{"View"}</a>
      ))
  }

  def filter(html:NodeSeq): NodeSeq = {
    Message.findAll(By(Message.author,S.params("author").head)).flatMap(
      message => bind("message", html,
              "title" -> Text(message.title.is),
              "author" -> Text(message.author.is),
              "view" -> <a href={"/blog/view/" + message.id.is}>{"View"}</a>
      ))
  }

  def view(html :NodeSeq): NodeSeq = {
    val id = java.lang.Long.valueOf(S.params("id").head).longValue
        Message.findAll(By(Message.id, id)) match {
      case msg :: Nil => bind("message", html,
        "title" -> Text(msg.title.is),
        "author" -> Text(msg.author.is),
        "date" -> Text(msg.date.toString),
        "text" -> Text(msg.text.is),
        "authorlink" -> <a href={"/blog/filter/" +
            msg.author.is}>{msg.author.is}</a>)
      case _ => Text("Invalid Message")
    }
  }

  def add(form: NodeSeq): NodeSeq = {
    val msg = Message.create
    def save(): Unit = {
      msg.save ; S.notice("Added "+msg.title); S.redirectTo("/blog/");
    }

    bind("message", form,
        "title" -> msg.title.toForm,
        "author" -> msg.author.toForm,
        "date" -> msg.date.toForm,
        "text" -> msg.text.toForm,
        "submit" -> submit("New", save))
  }

}
```

Listing 4.18: Blog Controler (*Snippet*)

Google Web Toolkit (GWT) [2] framework allows to create browser-based applications with a higher level of abstraction for web interfaces development. GWT main goals are to improve the development productivity and at the same time hide browser specific implementations [2, 30].

Development using GWT is made using Java[10] programming language and the code is compiled to JavaScript and optimized for several browsers (e.g., Internet Explorer, FireFox, Chrome, Safari, Android, iPhone) [2, 22]. The compiler also performs other optimizations like removing unused code, string optimizations, in-line methods, and move constant values to call places [2, 19, 30].

The compiler ensures that the application experience is the same using any browser by creating one application version for each browser and language combination (e.g., FireFox/English, FireFox/Portuguese, Chrome/Portuguese). Combinations can also be based on other parameters, defined by the developer (e.g., gender, age). During application boot, the correct combination is chosen for the user's browser and language and only the required version is loaded, decreasing the amount of data transfered across the Internet and it also ensures that the application will work as expected with the user's browser and language [2, 19, 30].

This kind of applications rely on asynchronous communications with the server. These communications are made using the XMLHttpRequest mechanism and the data exchanged can use JSON[11], XML, or a user defined protocol to communicate with the server, since the server side has a different language.

4.5 Programming Language and Database Integration

The usual architecture for applications with persistent data uses general purpose programming languages to define the computations and a database to access the application data [16]. This kind of applications is mainly developed using an object-oriented programming language and a relational database [16, 25]. The integration between these two layers is one of the most important aspects during application design, especially in web applications [17]. The relational model does not have a direct match with the available structures in a object-oriented programming language, requiring to use an intermediate language to perform the communication between each other. Instead of iterating data structures with loops and conditions, the communication with the database is usually done through SQL [16, 17].

The integration problem between relational databases and programming languages,

[10] http://www.java.com/
[11] JavaScript Object Notation

```
String empQuery = "SELECT e.name, e.salary, d.name as deptName"
    + " FROM (Employee e INNER JOIN Department d ON d.ID = e.department)"
    + " INNER JOIN Employee m ON m.ID = e.manager"
    + " WHERE e.name LIKE ? AND e.salary > m.salary"

Connection conn = DriverManager.getConnection(...);
PreparedStatement stmt = con.prepareStatement(empQuery);
stmt.setString(1, prefix + "%");
ResultSet rs = stmt.executeQuery(empQuery);
while ( rs.next() ) f
  print( rs.getString("name") );
  print( rs.getDecimal("salary") );
  print( rs.getString("deptName") );
}
```

Listing 4.19: JDBC Example [16]

also known as *impedance mismatch*, occurs due to the differences between the relational model and the object-oriented paradigm [16, 17]:

- **Type Mismatch.** Strings usually have limited length in databases, which is not the case in C#, VB.NET, or Java [25]. The precision of numeric values is another difference, but despite the difficulty to create a match between models, it is possible [16].
- **Data Models.** Concepts like inheritance are not directly supported by most relational databases (Figure 4.4) [25].
- **Programming Models.** Languages like C# or Java require loops and conditions to select data. Relational databases use a high-level declarative way to express the data we want.
- **Encapsulation.** In object-oriented languages, objects have behavior and data associated with them. Relational databases have a clear separation between data and behavior. It is only possible to act on database data through SQL queries or stored procedures.

One of the common approaches to integrate databases in programs, is to write SQL queries as strings inside the programming language code (Listing 4.19) [7, 16, 31]. This approach is highly error prone, since common errors such as typos in field names are only detected during runtime (typically in the test phase). A similar approach is to use a library of "prepared statements" which is safer but not very flexible, since the programmer has to adjust the library every time a change in the database is made [17].

A more evolved solution is to use an Object-Relational Mapping (ORM) interface where each object maps to a set of attributes in a database table. Relationships between

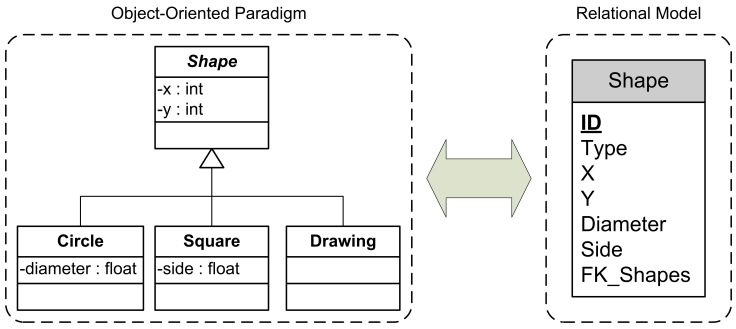


Figure 4.4: Object-Oriented Paradigm and Relational Model Mapping [25]

tables are represented using object references and n-ary relationships represented with collections of references [31]. This kind of interface offers an automatic mapping solution, but is not trivial, as shown in Figure 4.4. This figure shows the inheritance concept from the object-oriented paradigm that does not exists in the relational model. An extra attribute Type is added to represent the several types of objects (Circle, Square, and Drawing). This mapping also causes to have attributes that only belong to a certain type, to be available in all the others (Diameter or Side). Object-oriented languages primitive types ca not be null, but relational model supports null, and sometimes it may mean an unknown value (e.g., table joins).

All the approaches mentioned have advantages and drawbacks. Queries as strings are simple and fast to implement but do not allow any kind of static checks, while solutions like ORM (e.g., Hibernate) provide a safer solution to this problem, but are in many cases too heavy [25]. From these problems, we conclude that the best approach is to use a language that supports static checks between both layers. This requires the usage of a programming language with native support for queries [17, 31]. LINQ addresses the impedance mismatch problem by having built-in support for queries similar to SQL queries [10]. ScalaQL extends the Scala language using operator overloading, implicit conversions, and call-by-name semantics to implement an integrated query context [31]. In order to have a powerful integration between relational databases and programming languages it is necessary to avoid some of the problems that emerge. The best approach consists in integrating query support in the language itself, turning the compiler able to statically check queries for errors [17, 31]. As for our language we opted to have a native support for queries in our syntax (similar to LINQ) and use a type system with support for static checks for the entire application, including queries.

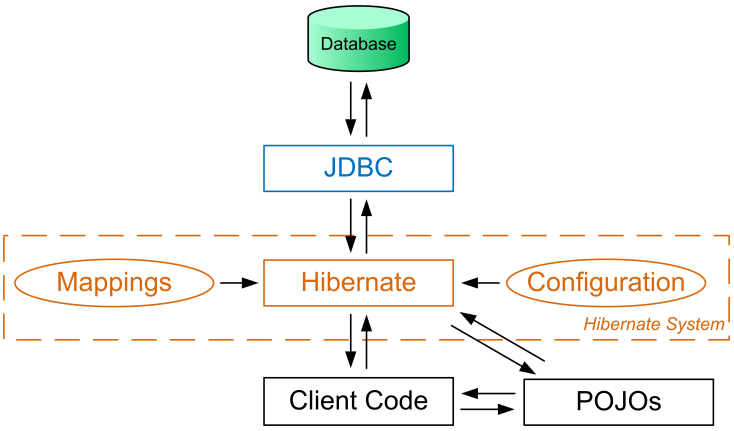


Figure 4.5: Hibernate Integration in an Application [24]

4.5.1 Hibernate

Hibernate [3] is an open source Object-Relational Mapping implementation designed to mediate the interaction between the application and a relational database [7].

Figure 4.5 shows how Hibernate is integrated in an application. Hibernate is incorporated between the Client Code and a JDBC[12] Driver. The purpose of a JDBC Driver, besides being responsible for the communication with the relational database, is to abstract the underlying connection details for each relational database, since each relational database supports a different set of features and different versions of SQL. To circumvent these differences, Hibernate abstracts DBMS into dialect classes where each supported database has its own dialect. Hibernate supports over 20 different dialects (e.g., MySQL, DB2, Oracle, Sybase, PostgreSQL) [3,7,24].

To use Hibernate the programmer defines persistent classes that are mapped to database tables. The mapping between persistent classes and database tables can either be done using Java annotations, or though a XML mapping file [7]. Inheritance relationships and various other relationships between classes are also supported by Hibernate [24].

Hibernate Query Language (HQL) allows us to request arbitrary information from the database. HQL is an object-oriented query language with its own syntax and grammar, similar to SQL, but instead of operating on tables and columns, it operates on persistent objects [24]. Although, queries are composed using string concatenation (Listing 4.20) therefore, type errors such as non-existing columns are only detected during runtime [35].

[12]Java Database Connectivity

```
String supplierHQL = "from Supplier where name='MegaInc'";
Query supplierQuery = session.createQuery(supplierHQL);
Supplier supplier = (Supplier) supplierQuery.list().get(0);

String hql = "from Product as product where product.supplier=:supplier";
Query query = session.createQuery(hql);
query.setEntity("supplier",supplier);

List results = query.list();
```

Listing 4.20: Hibernate Query Example [24]

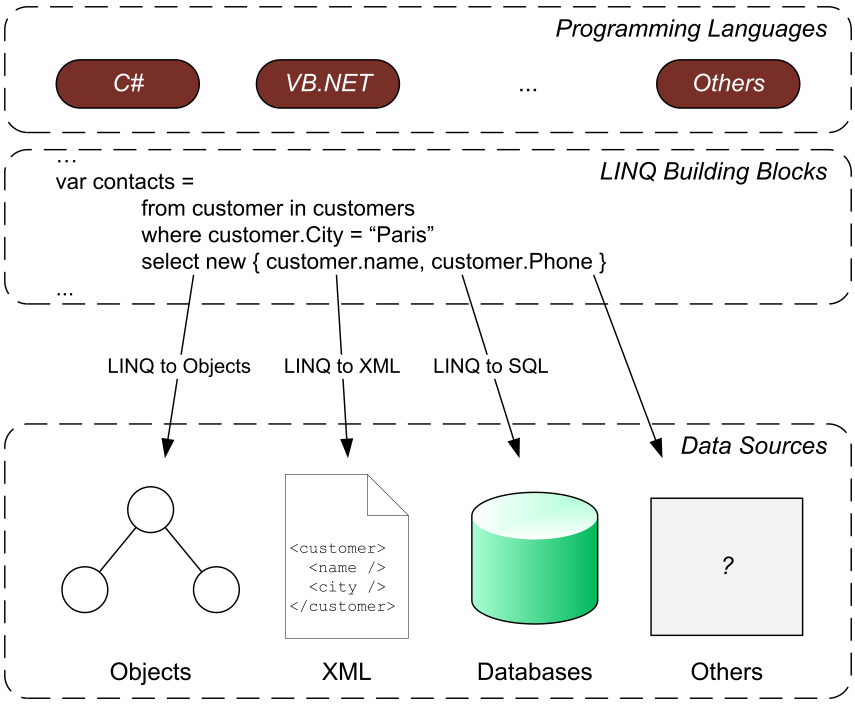


Figure 4.6: LINQ Architecture [25]

4.5.2 LINQ

As a way to address the impedance mismatch between the database layer and the programming language layer, Microsoft introduced the Language INtegrated Query (LINQ) component in the .NET platform [10].

LINQ offers a built-in support for queries, using a declarative syntax similar to SQL. This component allows to access in-memory collections like arrays or lists (LINQ to Objects), XML files (LINQ to XML), or relational databases (LINQ to SQL). It is also possible to create extensions to access other data sources (e.g., file systems, LDAP[13]) [17, 25].

[13]Lightweight Directory Access Protocol

```
val underAge = for {
  p <- Person
  c <- Company

  if p.company is c
  if p.age < 14
} yield p
```

Listing 4.21: ScalaQL Example [31]

```
SELECT p.*
  FROM people p JOIN companies c ON p.company_id = c.id
  WHERE p.age <14
```

Listing 4.22: SQL Query Obtained by ScalaQL From Listing 4.21

Using a common syntax to access different sources of data (Figure 4.6) makes possible to change between a database and XML without modifying the application code to use XQuery instead of SQL [25]. In addition to a common interface for several data sources LINQ also provides static checks during compile time.

4.5.3 ScalaQL

It is not always possible to integrate new syntax instructions in a language, like Microsoft has done with LINQ. ScalaQL language uses operators overloading, implicit conversions, and call-by-name semantics to extend Scala programming language, implementing statically checked queries by the Scala compiler itself [31].

ScalaQL transforms for-comprehensions (Listing 4.21) into sequences of flatMap, map, and filter method calls. For the example shown in Listing 4.21, the return type is Query[Person], when this type is implicitly converted, during execution, to a Seq[Person] type, the conversion to SQL is done, obtaining a query similar to Listing 4.22.

4.6 Links

Even with a server-side programming language that is able to statically check queries, like C# with LINQ or ScalaQL, the web interface is always designed with HTML or similar. With frameworks like GWT, communication between the interface and business logic layers is made using specific protocols (e.g., JSON, XML). The mismatches between the server and client sides can lead to other issues such as form input data validation. In order to ensure that the integration between the several interacting parts is not an application weakness, it becomes important to use a programming language

to design all three application layers.

Links [18] programming language solves the mismatches between layers by allowing the programmer to define entities (database tables), interface, and business logic using a functional and typed language with static checks and type inference.

The compilation process is done using a component written in OCaml. During compile time, the application code is split into the three common layers: interface, business logic and queries. The interface and business logic methods have a specific annotation in the language itself to identify to which layer they belong. The compiler submits every component through a series of static checks and the interface code is converted to HTML and JavaScript, queries converted to SQL, and the business logic remains in the server and is interpreted by the OCaml component. Although the interface definition mixes HTML code with Links code, thus not having a clear separation between layers.

Links authors argue that their implementation is scalable by preserving session state in the client [18]. On the one hand the server does not waste resources with session data but, on the other hand, every time a page is requested the session state is serialized and transfered between the client and the server and then back. This approach is more scalable in the sense that server resources are not used by users' data, but it has some security issues, transferring session data back and forth from the server, data like passwords, or critical user data may travel the Internet constantly.

The code fragment shown in Listing 4.23 implements a simple dictionary. This dictionary allows to view definitions and has a suggest feature that uses AJAX and a database table containing the definitions. The main page displays a simple form with a input text element that, when filled, displays the definition for the word written. This definition is done using plain XML with embedded code in curly braces and some special annotations like l:name. Business logic functions may also contain special annotations (client or server) to specify in which layer the function will be executed, e.g. functions that access database tables (completions) must be executed at the server whereas the suggest function that modifies the user interface must be executed in the browser. Links business logic functions may contain XML/HTML interface code, where in our language we create a clear separation between layers. Although Links already has support for features like AJAX that we consider important feature to develop richer web applications.

```
fun lowercase(s) {
  for (c <- s) [toLower(c)]
}

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

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

fun completions(pre) server {
  var wordlist = table "wordlist" with (
    word : String,
    type : String,
    meaning : String
  ) from (database "dictionary");
  if (pre == "") []
  else {
    query [10] {
      for (w <-- wordlist)
        where (w.word =~ /^{pre}.*/)
        orderby (w.word)
          [w]
    }
  }
}

var handler = spawn {
  fun receiver() {
    receive { case Suggest(pre) -> suggest(pre); receiver() }
  }
  receiver()
};

page
  <html>
    <head><title>Dictionary suggest</title></head>
    <body>
      <h1>Dictionary suggest</h1>
      <form l:onkeyup="{handler!Suggest(pre)}">
        <input type="text" l:name="pre" autocomplete="off"/>
      </form>
      <div id="suggestions"/>
    </body>
  </html>
```

Listing 4.23: Links Suggest Dictionary Example

4.7 WebDSL

WebDSL [35] is a domain specific compiled language dedicated to web applications development. The application code is split into the three layers and compiled to corresponding target languages. The interface layer uses JavaServer Faces (JSF) as target language and is concerned with generating web pages and interpreting user events. The database layer contains a relational database and a Object-Relational Mapping to take care of the communication with database and translate relational data into objects. The bridge between the interface and database layers is completed with Enterprise Java Beans (EJB3). Although the generated code is divided into three layers, WebDSL language combines the user interface with the business logic layer violating the Model-View-Controller [27] pattern, thus not providing a clear separation between layers in the application definition in WebDSL language.

Even using a statically typed language, which ensures that many common errors are caught at compile time, this does not ensures that web applications developed are error free. With JSF, web pages are only checked during runtime causing runtime exceptions such as missing or non-supported tags, references to non-existing properties and references to non-existing components. Seam framework, that combines JSF pages and EJB3, also has similar problems. Java Persistence API and Hibernate queries are composed using string concatenation, therefore, syntactic and type errors (e.g. non-existing columns) are only detected at runtime. While most of the times such errors are detected during the test phase, WebDSL tries to avoid this errors by statically checking programs (Listing 4.24). Hibernate Query Language is embedded in WebDSL syntax as expressions. With queries integrated in the syntax of the language, syntactic errors are caught at compile time by WebDSL compiler. The WebDSL type checker also checks the consistency of queries against the data model and local variable declarations. WebDSL compiler also performs static checks on application definitions, page navigation, and others.

4.8 Ur/Web

Ur/Web [13, 14] is a domain specific language for web applications development. The base language is Ur, which is a strongly statically typed and purely functional language based in ML and Haskell.

Ur/Web programming language allows to create dynamic web applications with persistent data stored in databases. With this language the programmer writes server-side code and client-side code in the same statically-typed language, and the compiler

```
entity User { name :: String }

define page user(u : User) {
  text(u.fullname)
  text(us.name)
  navigate(foo()){"bar"}
}

$ dsl-to-seam -i test.app
[error] entity 'User' has no property 'fullname'
[error] variable 'us' has no declared type
[error] link to undefined page 'foo'
```

Listing 4.24: WebDSL Compile Errors Example [35]

```
table t : { A : int }

fun list [u] (_ : fieldsOf u [A = int]) (title : string) (x : u) =
    xml <- queryX (SELECT * FROM x)
           (fn r : {X : {A : int}} => <xml><li>{[r.X.A]}</li></xml>);
    return <xml>
      <h2>{[title]}</h2>
      <ul>{xml}</ul>
    </xml>

fun main () =
    listT <- list "T" t;
    return <xml><body>
      {listT}
      <br/>
      <form>Insert: <textbox{#A}/> <submit action={ins}/></form>
    </body></xml>
and ins r =
    dml (INSERT INTO t (A) VALUES ({[readError r.A]}));
    main ()
```

Listing 4.25: Ur/Web Example

transforms the server-side code to native code and the client-side code to JavaScript. Database tables are defined inside Ur/Web modules, and the strongly-typed query library mimic a subset of SQL.

The Ur/Web type system ensures that an application does not suffer from code-injection attacks, executes invalid SQL queries, performs invalid marshaling or unmarshaling in communication with SQL databases or between the browser and the web server. An application can also use different Database Management Systems like MySQL, PostgreSQL, or SQLite and when compiled, a standalone web server is included.

Listing 4.25 shows a code fragment in Ur/Web language. This example allows to

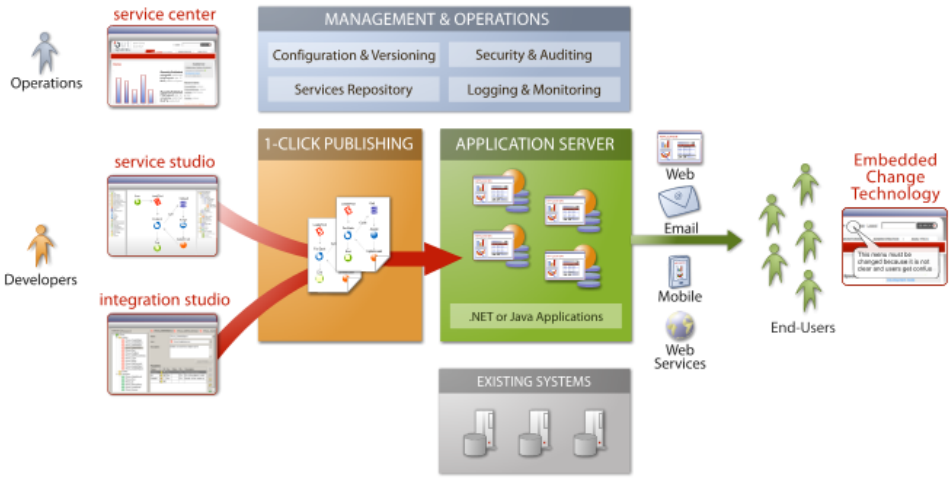


Figure 4.7: Agile Platform Architecture [4]

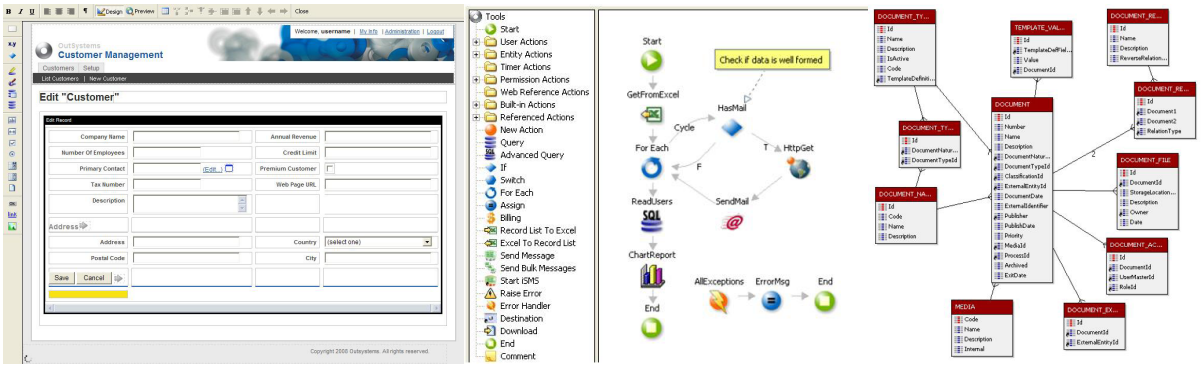
store simple text values in a database table (t) and view all existing values. The main screen is generated by the function main where it calls the list functions. This last function performs a database query against the table t selecting all fields and values, then for each value it displays an HTML <li> element and returns the entire list. The main function also has a simple form that allows us to insert new values, when submitted the function ins is executed and inserts the new value in the database table. Comparing to our approach, Ur/Web also allows to define all three common layers from a common web architecture, although it does not provide a clear separation between the business logic layer and interface layer, since functions may contain HTML and business logic, as shown in the example.

4.9 Agile Platform

Agile Platform from OutSystems [4] aims at solving the problem between the three layer integration, like Links, WebDSL, and Ur/Web. This platform uses a graphic environment to develop web applications and has four components (Figure 4.7): *Service Studio*, *Service Center*, *Integration Studio*, and *Embedded Change Technology*.

We will focus on the Service Studio component, since is the one that lies in the scope of this work. Service Studio allows to graphically define web pages, business logic, and the relational data model. Web pages are designed with a WYSIWYG[14] editor (Figure 4.8a). The application business logic is defined using *Action Flows* (Figure 4.8b), i.e., oriented graphs, possibly cyclic, where each node represents an operation. Available operations are, among others, conditional expressions (if, switch), SQL queries, or

[14] *What You See Is What You Get*



(a) Web Pages (b) Business Logic (c) Data Model

Figure 4.8: Service Studio

user defined actions. The third layer, relational data model, is also designed using a graphic editor (Figure 4.8c). Changes in the relational model are checked by Service Studio and, if necessary, modifications are recommended.

In addition to application development, Service Studio can also publish an application using the *1-Click Publishing* feature. This feature checks the integrity of the entire application, compiles, and publishes everything. The compilation process transforms the application definition to C# or Java. It is also possible to use Oracle or Microsoft SQL Sever as the Database Management System. During the compilation process, several optimizations are performed, such as query grouping and removal of duplicate queries in loops, decreasing the number of database accesses and improving application performance.

4.10 Discussion

Throughout this chapter we presented several tools related with the development of web applications. Ruby On Rails, CakePHP, and Scala Lift frameworks are similar, in the sense that they all use abstractions to access the database layer and the application is defined in a Model-View-Controller [27] pattern. For each framework, we present the same example to serve as a mean of comparison between each framework and our language, where we also present a similar example. Both Rails and CakePHP use interpreted languages (Ruby and PHP), so neither one of them supports static checks. This gap is one major difference when compared to our language, where we designed a language with support for static checks, like Scala Lift. Another difference is the usage of a specific querying language. All three frameworks have their own querying language, we opted to use a common querying syntax in our language (similar to LINQ and SQL). In our approach the interface is defined using blocks from the language

syntax while these three frameworks use standard unverified HTML/XML.

Google Web Toolkit addresses the problem in the interface layer, i.e., the mismatch between browsers. Since this work is focused on the integration between all three common layers and currently does not have support for features like AJAX (heavily used in applications developed with GWT), we present GWT framework just as a study for future work.

The integration between programming languages and databases has several mismatches, and from our readings we consider that integrating query support in the language itself, turning the compiler able to statically check queries for errors, is the best approach. We have chosen a syntax similar to LINQ (based on SQL), since our business logic language fragment is more close to the imperative paradigm, in comparison to the others approaches like for-comprehensions that are more common in functional languages.

We also describe Links, WebDSL, Ur/Web, and Agile Platform that aim at solving the integration between the three common layers. While these frameworks have a compile step, we opted to use an interpreted language with static type checks. By using an interpreted language we managed to use a dynamic reconfiguration mechanism in our prototype that is able to update an application definition without needing to restart the application. Both Links and Ur/Web do not provide a clear separation between the business logic layer and the interface layer, while in our approach we have two different language fragments for actions and screens. In Links, business logic may contain XML/HTML code whereas in Ur/Web the interface is defined along side with the business logic code. Unlike WebDSL, our type system checks all fields used in queries (including **select** and **where** clauses) detecting errors like invalid field names.

5

Final Remarks

This MSc work main goals were to design a simple and small core language for web applications, and implement a prototype that allows us to define applications using a web based development environment. Our language offers an integrated programming environment that allows to define the interface, business logic and database manipulation operations. Using a higher level of abstraction in our language, in comparison to general purpose languages, we avoid the several existing mismatches between layers. These mismatches include unverified code between the client and server sides, and the SQL code within the application and the database. Using static verifications within all application code, including communication code between layers, we provide basic safety of programs and elimination of common programming errors.

Our prototype implementation provides a small but effective web based development environment where users can act directly over the application definition and update it. This prototype includes a dynamic reconfiguration mechanism that ensures that applications modifications are put into practice without needing to restart the application.

This language also aims at potentiating the verification of other more sophisticated properties, in particular, we refer to properties related to data security and access control [9, 26, 33] and related to the coordination of several interacting parts in distributed systems [34]. The language is already being extended to demonstrate security related

properties checking by means of refinement types [9,21].

As the result of the entire language design and implementation we consider that the final result reached our main goals of creating a simple and small language with a working prototype that will allow further studies of properties related with web applications like data security and access control.

5.1 Future Work

Even with an extension already in progress we consider that are still some features that can improve or make our language and prototype more complete.

Nested Queries. In most cases it is possible to circumvent this gap by executing multiple queries and using iterators, but a feature like this makes a language more expressive.

Query Optimization. In web applications it is common to repeat queries inside iterators and most of the times it is possible to optimize those queries to avoid multiple requests to the database.

Lazy Query Evaluation. Currently when a **from** query is executed, all data is fetched from the database and placed in memory We could avoid this by using lazy evaluation and only fetch the results as they were needed.

***Delete* Operation and *Order By* Clause.** Entities can be manipulated through the **insert**, **update** and **from** expressions, however the *delete* expression is not yet included in our language syntax. The same happens with the *order by* clause in **from** expressions, since we currently cannot order query results.

User Interface. The current number of interface blocks do not consider some HTML elements such as radio buttons, check boxes, or text areas. Still related with screens, we cannot create a screen by composition, i.e., create a screen based on several defined screens, something that would offer even more modularity to the language.

Asynchronous JavaScript and XML. Adding the possibility to execute callbacks expressions through Asynchronous JavaScript and XML (AJAX) is something that could also be added to our language, allowing to develop richer web applications since this kind of applications are becoming more popular.

Sessions and Cookies. Management of session variables and user cookies is currently not part of our language, but features like this are highly used in web applications to store temporary and specific data about each user. This feature would make our language more usable in practical scenarios.

Bibliography

[1] CakePHP Manual: The Cookbook, Jan 2010. http://book.cakephp.org/.

[2] Google Web Toolkit, Jan 2010. http://code.google.com/webtoolkit/.

[3] Hibernate, Jun 2010. http://www.hibernate.org/.

[4] OutSystems, Jan 2010. http://www.outsystems.com/.

[5] Ruby On Rails, Jul 2010. http://www.rubyonrails.org/.

[6] Ahsanul Bari and Anupom Syam. *CakePHP Application Development*. Packt Publishing, 2008.

[7] Christian Bauer and Gavin King. *Hibernate in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[8] Tim Berners-Lee. Information Management: A Proposal. *European Particle Physics Laboratory (CERN)*, March 1989.

[9] Luís Caires, Jorge A. Perez, João C. Seco, and Hugo T. Vieira. Refinement Types for Database Access Control. Technical report, UNL-DI-3-2010, Departamento de Informática, FCT/UNL, 2010.

[10] C. Calvert and D. Kulkarni. *Essential LINQ*. Addison-Wesley Professional, 2009.

[11] Kai Chan, John Omokore, and Richard Miller. *Practical CakePHP Projects*. Apress, Berkely, CA, USA, 2008.

[12] Derek Chen-Becker, Tyler Weir, and Marius Danciu. *The Definitive Guide to Lift: A Scala-based Web Framework*. Apress, Berkely, CA, USA, 2009.

[13] Adam Chlipala. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. *Programming Language Design and Implementation (PLDI) 2010, SIGPLAN Notices*, 45(6):122–133, 2010.

[14] Adam Chlipala. *The Ur/Web Manual*. Mar 2010. http://www.impredicative.com/ur/.

[15] The Web Application Security Consortium. Web Application Security Statistics, 2008. http://projects.webappsec.org/Web-Application-Security-Statistics.

[16] William R. Cook and Ali H. Ibrahim. Integrating Programming Languages and Databases: What's the Problem? In *ODBMS.ORG, Expert Article*, 2005.

[17] Ezra Cooper. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. *Database Programming Languages (DBPL) 2009, LNCS*, 5708, 2009.

[18] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. *Lecture Notes in Computer Science*, 4709:266–296, 2006.

[19] Ryan Dewsbury. *Google Web Toolkit Applications*. Addison-Wesley, 2008.

[20] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly, 2008.

[21] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[22] Robert Hanson and Adam Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.

[23] Patrick Lenz. *Simply Rails 2*. Sitepoint, Collingwood, Vic. :, 2nd ed. edition, 2008.

[24] Jeff Linwood and Dave Minter. *Beginning Hibernate, Second Edition*. Apress, Berkely, CA, USA, 2010.

[25] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *LINQ in action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[26] Mário Pires and Luís Caires. A type system for access control views in object-oriented languages. *Foundations and Applications of Security Analysis, Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, 2010.

[27] Trygve Reenskaug. Models - Views - Controllers. *Technical note, Xerox PARC*, December 1979.

[28] David Robinson and Ken A.L. Coar. The Common Gateway Interface (CGI) Version 1.1. RFC 3875 (Informational), October 2004.

[29] Sam Ruby, Dave Thomas, and David Hansson. *Agile Web Development with Rails, Third Edition*. Pragmatic Bookshelf, 2009.

[30] Bram Smeets, Uri Boness, and Roald Bankras. *Beginning Google Web Toolkit: From Novice to Professional*. Apress, Berkely, CA, USA, 2008.

[31] Daniel Spiewak and Tian Zhao. ScalaQL: Language-Integrated Database Queries for Scala. *Software Language Engineering 2009, LNCS*, 5969:154–163, 2010.

[32] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[33] Bernardo Toninho and Luís Caires. A spatial-epistemic logic and tool for reasoning about security protocols. Technical report, Departamento de Informática, FCT/UNL, 2009.

[34] Hugo T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service oriented computation. *European Symposium on Programming (ESOP) 2008, LNCS*, 4960, 2008.

[35] Eelco Visser. WebDSL: A case study in domain-specific language engineering. *Generative and Transformational Techniques in Software Engineering (GTTSE 2007), LNCS*, 5235:291–373, October 2008.