LiveWeb – Core Language for Web Applications

Miguel Domingues and João Costa Seco

CITI - Departamento de Informática FCT/UNL, Lisboa, Portugal
miguel.domingues@fct.unl.pt, joao.seco@di.fct.unl.pt

Abstract. We present a typed core language for web applications that integrates interface definition, business logic and database manipulation. By expressing the interactions between different application layers in the same programming language, we gain the benefits of strongly typed languages without loosing the programming flexibility of interpreted languages which is frequently an argument in favor of unsafe programming languages. We describe a prototype of a programming environment and runtime support system for our language that allows a very dynamic style of web application development.

Keywords: Web applications, programming languages, type systems

1 Introduction

The main-stream of web application development is usually 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, the application logic is specified using general purpose programming languages to define computations and specialized query languages to access the information stored in databases. The two language paradigms have several mismatches [5,6] making the integration between layers one of the most important aspects of web application development. Typically, layers interact through dialects and programming conventions, and communication code is not subject to effective mechanical verification and is highly error prone. Writing SQL queries as strings is a simple and fast way to implement database applications but doesn't allow for any kind of static checks. Object-Relational Mapping approaches provide a safer solution to this problem but are in many cases considered too heavy [11].

Furthermore, web application development is very high demanding for rapid construction and constant change, which gave rise to a series of flexible languages that trade the benefits of statically strongly typed programming languages for the advantages of dynamically typed interpreted languages (e.g., PHP, ASP, Ruby). Some development frameworks targeting web applications (e.g., Ruby On Rails, CakePHP) provide scaffolding features to increase developers' productivity, others provide extensions to general purpose languages and include typing for database operations [3,7], a third category of frameworks choose to use domain specific languages to provide program safety by construction [1,4,7,14]. We

argue that the latter approach, to integrate query support in the language and hence enable static verification between layers, has clear advantages and is more challenging from a programming language perspective.

Although the rising of 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 [2,10,12] and related to the coordination of several interacting parts in distributed systems [13]. In order to allow future experiments on these theoretical studies on type systems we introduce a typed core language for web applications that integrates the typing of interface definition, business logic and database manipulation. A more complete description of the language is available in [8]. Our approach compares to Links [7] and Ur/Web [4] that also define strongly typed languages but we take a more limited approach of starting from first principles with primitive operations, types and a clear separation between interface and program, and thus allowing for formal studies to be easily applied here.

We also describe an implementation of an interpreter and a highly flexible programming environment for our language, designed to provide a dynamic programming style where the developers act directly over the actual running code without loosing the global integrity checks of interpreted languages.

2 Core Language for Web Applications

Our core language has three main 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 database 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 now illustrate the syntax of the language by means of the code fragment in Fig. 1 implementing a phone number directory. We define an entity called Person containing phone numbers and names by enumerating its attributes and corresponding types. The interface of the application is defined in a screen called directory, built by iterating the results of a **from** expression (written in a syntax similar to LINQ [3]) that fetches all values stored in entity Person. The language fragment used to define web pages is inspired in the nested structure of web page blocks, it contains an **iterator** expression that maps query results in web page blocks, and also contains input elements (**textfield**) and actuator elements (**button**). Input elements declare local variable names that can be used in expressions that pass the control flow from the browser back to the web server. The **button** element in screen directory calls action addPerson using as arguments the values given by the user in the text fields, which are available through to the local names name and phone. Action addPerson adds a new row to entity

```
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(nm:String, ph:String):Block {
  insert { name = nm, phone = ph } in Person;
  directory()
}
```

Fig. 1: LiveWeb Example

Person with the given values for name and phone. After the insertion of a new row, screen directory is rendered in the browser. The type system of the language ensures that there are no runtime errors due to ill-formed queries, with missing entity names or ill-typed arguments in where clauses, it ensures that screens are rendered properly, e.g. with no missing information from entity attributes, and that all actions used in screens exist and expect matching parameter types, etc.

3 Runtime Support System

We implement our language in a runtime support system that combines a web server with a language interpreter and a database for application data. The system also provides a wiki style development environment based on a persistent and versioned code base. Source code is stored, versioned, and organized in a database instead of being scattered in files. This allows for the type safe dynamic reconfiguration of the system, since we maintain as active the code that was last verified as well typed. The UI fragment evaluates to regular HTML code with JavaScript for name binding and activating continuation code in the server.

Our runtime support system provides two functioning modes, one for executing the application and another for editing and checking the source code. The first mode of interaction, the *execution mode*, allows for actions and screens to be called by standard URL conventions using the name of element (action or screen) and by indicating the corresponding arguments by means of literals. More complex browser interactions can be achieved by standard techniques but are out of the scope of this work. Query expressions are evaluated to regular SQL expressions and executed in the database.

The *development mode* is also available through the browser (usually through an "edit" button or link). It lets the user access and change all available elements of an application. The runtime support system stores screen, action, and entity definitions as separate pieces of code, and establishes a notion of published version of an application built from the latest verified copies. When the structure

of an entity is modified, the database model of application data must also be modified to match the new entity definition. For the sake of simplicity, entity data is transformed in the more direct way in order to keep applications working.

4 Final Remarks

On the one hand, this work aims at developing a simple and small language that could be easily extended and allow the formal study of type related properties like data security and access control. On the other hand, it aims at providing an implementation of a runtime system for the language that works like a workbench for those extensions. Security related property checking techniques based on refinement types [9] are presented in [2] and a prototype is already available from the authors' web page. There are many technological issues that can be improved to make the language and runtime system more usable (e.g. nested queries, lazy query evaluation, asynchronous page updates, etc.) and robust (keeping versions of data of deleted columns, etc.). However, the main goal is to provide a framework to future experiments on type systems for web applications.

Acknowledgments. This work is partially supported by the Certified Interfaces project NGN44-CMUPortugal. We thank to Luís Caires, António Melo and Lúcio Ferrão for the discussions at OutSystems that motivated this work.

References

1. OutSystems (Jan 2010), `http://www.outsystems.com/`
2. Caires, L., Perez, J.A., Seco, J.C., Vieira, H.T.: Refinement Types for Database Access Control. Tech. rep., UNL-DI-3-2010, Dep. Informática, FCT/UNL (2010)
3. Calvert, C., Kulkarni, D.: Essential LINQ. Addison-Wesley Professional (2009)
4. Chlipala, A.: Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. PLDI 2010, SIGPLAN Notices 45(6), 122–133 (2010)
5. Cook, W.R., Ibrahim, A.H.: Integrating Programming Languages and Databases: What's the Problem? In: ODBMS.ORG, Expert Article (2005)
6. Cooper, E.: The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. DBPL 2009, LNCS 5708 (2009)
7. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. FMCO 2006, LNCS 4709, 266–296 (2006)
8. Domingues, M., Seco, J.C.: Definition of a Core Language for Web Applications (LiveWeb). Tech. rep., UNL-DI-4-2010, Dep. Informática, FCT/UNL (2010)
9. Freeman, T., Pfenning, F.: Refinement Types for ML. PLDI 1991, SIGPLAN Notices 26(6) (1991)
10. Pires, M., Caires, L.: A type system for access control views in object-oriented languages. In: ARSPA-WITS 2010. LNCS (2010)
11. Spiewak, D., Zhao, T.: ScalaQL: Language-Integrated Database Queries for Scala. SLE 2009, LNCS 5969, 154–163 (2010)
12. Toninho, B., Caires, L.: A spatial-epistemic logic and tool for reasoning about security protocols. Tech. rep., Dep. de Informática, FCT/UNL (2009)
13. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service oriented computation. ESOP 2008, LNCS 4960 (2008)
14. Visser, E.: WebDSL: A case study in domain-specific language engineering. GTTSE II, LNCS 5235 (2008)