A Core Language for Data-Centric Processes

Luísa Lourenço

CITI e Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

Abstract. In the past decades, with the proliferation of the Internet, data-centric applications have become widely used. These applications deal with distributed information that can be kept in a repository that is shared among multiple participants with specific roles; or with shared, but not necessarily distributed, information that can be manipulated in a collaborative multiuser environment.
In order to model and/or develop data-centric programs, several proposals were made in the past years that focus mainly on Data Manipulation Languages (DML), while leaving other interesting scenarios behind. For instance, often organisations express processes via workflows that describe all the steps necessary to complete them.
In this paper, we propose a core language to express such processes by presenting a typical DML core augmented with primitives to describe and create tasks/workflows. In our proposal, these new entities are first-class values, thus enabling us to talk about richer scenarios where we can have, for example, pending tasks stored in a database in order to be dealt with later.

Key words: Data-centric applications, Programming Languages, Data Manipulation Languages, Workflows

1 Introduction

In the past decades, with the proliferation of the Internet, data-centric applications have become widely used. These applications deal with distributed information that can be kept in a repository that is shared among multiple participants with specific roles; or with shared, but not necessarily distributed, information that can be manipulated in a collaborative multiuser environment.

In order to model and/or develop data-centric programs, several proposals were made in the past years that focus mainly on Data Manipulation Languages (DML) whose goal consist in interacting with data structured in a relational schema. This schema establishes relations between abstract entities which in turn represent collections of data. The most popular DML is undoubtedly IBM's Structured Query Language (SQL) [1,3], developed in the early 70s (when it was named Sequel). SQL allows us to create relations and then manipulate them by issuing queries over the relation to retrieve information from it. Moreover, SQL expressions can also perform changes over relations such as updates, deletion, and insertion.

On the other hand, often organisations express their processes via business processes that describe all the activities necessary for its completion. So business process management [8] (BPM) is fundamental in any organisation in order to optimise parameters such as the production costs and productivity as well as to assign workers to tasks. The supporting technology to automate a BPM is known as workflow, so a workflow is a tool to model processes by defining a set of tasks that must be fulfilled in a certain order by certain individuals of the organisation. Furthermore, it is often the case that these workflows are information intensive. For instance, a project management system must deal with high volume of data that is kept in databases, so a project submission process, in that system, must interact with the databases in order to: keep the submission information until it is later evaluated by a system's clerk; retrieve the necessary information to assess the viability of the submission; update the status of the submissions after its evaluation. Therefore, a model that integrates DML with workflows would allow us to reason about these data-centric systems.

2 Motivation

In this paper we propose a core language with constructions for data manipulation as well as for the definition, execution, and composition of tasks and workflows. While there has been much work done with respect to programming languages based on DML and specification languages for business processes, to the best of our knowledge, there is none regarding programming languages that integrate the two concepts together.

We find that a computational model based on tasks (activities in a business process), coupled together with data manipulation primitives, would further enrich typical data-centric scenarios. Like, for example, a project management system where we can model a submission process that suspends incoming submissions (encoded as tasks) until they are evaluated by a clerk. But more importantly, in the point-of-view of executable business processes, it is also our conviction that it is crucial to abstract about the data that is manipulated throughout the execution of the processes. This is due to the fact that it would allow for an analysis of such processes with regard to the information they yield and have access to, thus ensuring guarantees like data security.

The language we propose aims to provide a computational model centred on tasks and data, in order to study and reason about information security (where a task is in itself data). In general, it is possible with this language to statically analyse and verify properties about the correctness of the programs such as security properties (for e.g., data confidentiality and data integrity), type safety, as well as any other property that can be verifiable by a static analysis. Therefore, it is not our goal to propose a language for programming or modelling of real world applications but to provide with a model for analyses of information in the context of business processes.

To fulfil this goal, our proposal consists in three distinct aspects: a core of typical imperative primitives; a small set of DML constructions based on SQL expressions; and primitives to define, compose and invoke tasks/workflows. Since we are interested in scenarios where tasks can be kept in data storage, then our language will have tasks as a first-class value. We also believe that a task is a good abstraction for computational processes that can be suspended, resumed and stored in a repository/data structure.

Our objective in this paper is to discuss and motivate our preliminary views on a few important research issues and directions, as well as to advance the basic structure of a programming model for process based data centric software systems, which may conveniently support sophisticated verification of correctness and security. Ultimately, our long term goal is to contribute to the development of improved programming tools for the construction of more robust and more secure data-centric, process oriented, information systems.

3 Language

The syntax of our core language is given by the grammar in Figure 1 where we assume an infinite set of *names* (ranged over $n,\ m,\ \ldots$) and of *variables* (ranged over $x,\ y, \ldots$). The fragment e corresponds to the language's basic core: variables, values, field access $e.m$, application $e_1(e_2)$, functions' declaration $\lambda x.e.$, constants and variables' declaration, conditional, while loop, assignments, a primitive to obtain the head of a collection, and primitives for data creation and manipulation. Data manipulation primitives are the expected: creation of a relation with a set of attributes, **entity** $\mathrm{n}(m_1,\ \ldots,\ m_n)$; projection of a set of attributes in a relation under a given condition, **select** m_i **from** e_1 **where** $(m_j == e_2)$; insertion of an element on a relation, **insert** e_1 **in** e_2; update of a set of elements on a relation if a given condition is met, **update** m_i **in** e_1 **where** $(m_j == e_2)$ **with** e_3; and deletion of a set of elements on a relation if a given condition is met, **delete in** e_1 **where** $(m_j == e_2)$.

In fragment t we have primitives related to tasks: task definition Σ_i **def** $\mathrm{op}_i(\overline{x}) = t$ **in** t, that defines a set of tasks; task invocation, $\mathrm{op}(\overline{e})$; a task constructor, **let** $\mathrm{n} = \{t\}$ **in** t, to define tasks; and a set of primitives to compose tasks inside a task definition ($t; t$, $t \mid t$, t^*, and $\mathrm{op}(\overline{\textbf{in: } e}, \overline{\textbf{out: } x})$). The difference between a task definition and a task constructor lies, essentially, in the semantic: the former will only be executed upon invocation while the latter does not require an invocation to be executed. With task composition primitives we can define workflows by specifying which tasks are executed, in what order (either sequentially or in parallel), and what is the input and output of each task. For instance, the following code

```
def submitProject() = {
  prepareSubmisison(out: form);
  submit(in: form, out: msg);
}
```

$p ::= e$	program
$e ::= e_1; e_2$	sequence
$\mid e.m$	field access
$\mid$ **let** $x = e_1$ **in** e_2	let
$\mid$ **var** $x = e_1$ **in** e_2	var
$\mid e_1 := e_2$	assignment
$\mid \lambda\overline{x}.e$	abstraction
$\mid e_1(e_2)$	application
$\mid$ **if** e **then** e_1 **else** e_2	conditional
$\mid$ **while** e_1 **do** e_2	while
$\mid e.$**head**	get collection's head
$\mid$ **entity** n $(m_1, \ldots, m_n)$	create entity
$\mid$ **select** m_i **from** e_1 **where** $(m_j == e_2)$	select
$\mid$ **insert** e_1 **in** e_2	insert
$\mid$ **update** m_i **in** e_1 **where** $(m_j == e_2)$ **with** e_3	update
$\mid$ **delete in** e_1 **where** $(m_j == e_2)$	delete
$\mid x$	variables
$\mid t$	tasks
$\mid v$	values
$t ::= \Sigma_i$ **def** $\mathrm{op}_i(\overline{x}) = t$ **in** t	task definition
$\mid \mathrm{op}(\overline{e})$	task invocation
$\mid$ **let** n $= \{t\}$ **in** t	task constructor
$\mid t; t$	task sequence
$\mid t \mid t$	task parallel
$\mid t^*$	task recursion
$\mid \mathrm{op}(\overline{\textbf{in: } e}, \overline{\textbf{out: } x})$	task composition
$\mid \{e\}$	expression
$v ::= \{v_1, \ldots, v_n\}$	collections
$\mid [m_1 : v_1, \ldots, m_n : v_n]$	records
$\mid \langle\overline{\mathrm{op}(\overline{a})}\rangle$	task
$\mid$ **true**	true
$\mid$ **false**	false
$\mid ()$	unit

Fig. 1: Syntax

declares a task named submitProject whose definition is a sequential composition of tasks (so in fact submitProject is a workflow). Moreover, the definition states that the first task executed, upon invocation of the workflow, is prepareSubmission which will take no arguments and will deliver a result in the variable form; then this value is transmitted to the next step of the workflow, which is task submit that takes as argument the variable form and outputs a result that will be saved in the variable msg. After all the tasks are completed, the final result of the workflow will be saved in variable msg.

Values include booleans, unit value $()$, collections of values $\{v_1, \ldots, v_n\}$, records $[l_1 : v_1, \ldots, l_n : v_n]$, and tasks $\langle\overline{\mathrm{op}(\overline{a})}\rangle$. A task value is a reference to either a constructed or invoked task that can either be running or terminated. If it is still running then the reference will list all the possible actions (operations currently available through a task definition) and allow invocations on those actions. Let us take as an example the following task definition

```
def submit(doc) = {
  let t = {
     let id = doc.id;
     let backgroundInfo = getInfo();
     {
       def backgroundInfo() = { ... }
       +
       def accept() = { ... }
       +
       def reject() = { ... }
     }*
  };
  let s = [ submission_id = doc.id, submission = t ];
  insert s in submissionsDB;
  "Submission successfully completed!"
}
```

In this code snippet we use a task constructor to create a task with identifier t, which is afterwards kept in a record that is inserted in a relation named submissionsDB. Recall that a task created with a constructor does not need to be invoked, meaning it will be executed right away. Then, since there is a block of code with a set of tasks' definitions, the task will await for invocations of the three available operations: backgroundInfo, accept, and reject. So identifier t will have a reference that list all three tasks and allow us to invoke any of them.

4 A Project Management Workflow Example

We now present a toy example of how to use our language to specify a rich scenario of a project management service that offers two functionalities: project submission and checking a project's status. These functionalities can be encoded into two distinct workflows, submitProject and viewStatus, respectively, where each one describes the necessary steps for its completion. Moreover, a database of all submissions is kept in a relation named submissionsDB.

```
let submissionsDB = entity( submission_id, user, buzz, status, submission );
```

When a system's user invokes the viewStatus workflow, he will pass the identifier of the project he wants to check.

```
def viewStatus(id) = {
  (select status
   from pendingSubmissions
   where submission_id = id).head;
}
```

Then on the server's side, a query is executed on the submissions' database to select the status of the required project. The head of the result of the query (since the query returns a collection with one element) is then returned to the client and the workflow terminates.

To submit a project, a user will invoke the submitProject workflow. This workflow states that, in order to submit a project, it will require that the task prepareSubmission be executed first, followed by the task submit (that will take as input the result of the first task). The entire workflow is executed on the server's side, as expected.

```
def submitProject() = {
  prepareSubmisison(out: form);
  submit(in: form, out: msg);
}
```

So as soon as the user invokes the submitProject workflow, the prepareSubmission task will start executing and a reference (a task value) to the workflow is returned to the user.

```
def prepareSubmission() = {
    var form = [id = −1, user = "", buzz = ""];
    def fillForm(username, buzz) = {
      form.id = 123;
      form.user = username;
      form.buzz = buzz;
    };
    def submitForm() = { form; }
}
```

This task will create an empty form and await for the user to fill the form. Interactions with the user are performed via invocation of functionalities represented by subtasks inside an executing task to which the user has access to. Therefore, after generating the form, the task will suspend its execution and await an invocation to the subtask fillForm from the user. At that point, the user's workflow reference will list the fillForm subtask as an available task for invocation. The user fills in each form's field by invoking the fillForm subtask with his username and buzz information, and only then the task prepareSubmission resumes its execution and fills the form with the user's data and a form identifier. After the form is filled, the task will suspend its execution again to await for a submit request by the user (encoded as a subtask that the user must invoke in order to finish the submission of the form). When the user does so, the submit subtask will terminate by returning the filled form to the prepareSubmission task, which, in turn, will output the form to the submitProject workflow. The first step of the workflow is then completed and its output is passed to the second step, task submit, as input.

```
def submit(doc) = {
  let t = {
      let id = doc.id;
      let backgroundInfo = getInfo();
      {
        def backgroundInfo() = { backgroundInfo; }
        +
        def accept() = {
          update status in submissionsDB
          where submission_id = id with "Accepted";
          "Project accepted."
        }
        +
        def reject() = {
          update status in submissionsDB
          where submission_id = id with "Rejected";
          "Project rejected."
        }
      }*
    };
    let s = [ submission_id = doc.id,
              user = doc.name,
              buzz = doc.buzz,
              status="pending",
              submission = t ];
    insert s in submissionsDB;
    "Submission successfully completed!"
  }
```

When task submit starts, a task is created and (its reference) saved in a constant named t. The task is executed right away, initialising two local constants with information regarding the submission id (taken from the form), and the user's background information (represented as a call to a function named *backgroundInfo*), offering afterwards three functionalities (since we are not using sequential composition but instead a sum of tasks) to whoever has access to the constructed task: a subtask to obtain the user's background information, a subtask to accept the submission, and a subtask to reject the submission. A key point in this task, in comparison with the previous prepareSubmission task, consists in the visibility of these subtask's definitions. In this case, since we are using a task constructor, the subtasks are only visible to whoever has access to constant t, and so they are not associated with the submitProject workflow reference that the user holds. So, at this point, the constructor task suspends its execution and the execution control is given back to the task submit, who will create and insert a submission entry for the submission's database that will include all the information regarding the submission: submission identifier, user who submitted the project, the submission's buzz, submission's status and, finally, the constructed task t that offers the functionality necessary to assess the viability of the submission and consequent acceptance/rejection. Lastly, the submit task will return to the workflow a message stating that the submission was successfully completed, which in turn will output the message to the client (since it was the last step of the workflow, then its output will be the workflow's output). So this concludes the submitProject workflow, invoked by a system's user.

In order to have a submission evaluated, a clerk (from the project management system) will have to treat the submission: evaluate and then make a decision. We now show the workflow that treats a submission by a clerk.

```
def treatPendingSubmissions() = {
  getSubmission(out: submission);
  evaluateSubmission(in: submission, out: info);
}
```

In this workflow, the first step consists in retrieving a pending submission from the database and, afterwards, evaluate the submission. Thus, when the clerk invokes the workflow, he will receive a reference to the workflow and, on the server side, the steps necessary for its completion will be executed.

```
def getSubmission() = {
  var i = 0;
  var p = (select * from pendingSubmissions
                    where submission_id = i).head;
  while(p.status != "pending") do {
    i := i + 1;
    p := (select * from pendingSubmissions where submission_id = i).head;
  }
  p
}
```

The first step is the execution of task getSubmission to obtain a pending submission. This is achieved by querying the database until a pending submission is found, which will be the result of the task. The submission is then passed to the next, and last, task of the workflow.

```
def evaluateSubmission(s) = {
  if (evaluate(s.buzz)) then
    s.submission.accept();
  else s.submission.reject();
}
```

The last step of the workflow evaluates the submission obtained in the first step and outputs the result to the clerk that invoked the workflow. In this task, named evaluateSubmission, a function *evaluate* is used to assess the submission's viability. If viable, then the task that is stored in the retrieved submission entry will be used to change the submission's status to accepted. Otherwise, the task will be used to change the status to rejected. Recall that this is possible because we stored a suspended task in a database entry. So when the system retrieves the task from the database, it can resume its execution by either invoking the backgroundInfo subtask or the accept subtask or the reject subtask.

5 Related Work

Undoubtedly, SQL is the most well-known and widely used DML. In SQL, database tables are relations associated with a unique name, obeying a relational schema defined by a set of attributes and their domains. In SQL, command **create table** r(A_1D_1, …, A_nD_n) creates a relation r with attributes A_1, …, A_n whose domain is D_1, …, D_n, respectively. An SQL expression enables us to manipulate relations. For example, an expression to retrieve information from relations has the form **select** A_1, …, A_n **from** t_1, …, t_m **where** C that projects the attributes A_1, …, A_n from the cartesian product of relations t_1, …, t_m if the predicate C is true. Moreover, an SQL expression can also perform changes over relations such as updates, deletion, and insertion.

In the past years, SQL has inspired several proposals of programming languages that integrate data operations into applications. For instance, LINQ [2], is a .NET framework that extends C# with native query operations such as projection, filter data, and aggregate data. These operations can be applied not only to relations (relational databases) but also arrays, enumerable classes and XML. Most of the ideas presented in LINQ are the result of previous work [6].

Another relevant work is Links [7]. It incorporates query-like expressions that are later converted into SQL. This work focus on web applications' programming and proposes to integrate all the three tiers that are typically found in this setting: the web browser, the web server, and the back-end systems (like databases). This is achieved by having the their system generate code for each tier: Javascript for the browser, bytecode for the server, and SQL for the database.

More recently, there is Dminor [5], a functional language (a subset of Microsoft's M modelling language [4]) with a small set of data manipulation primitives such as an accumulate expression over collections and addition of an element to a collection. This small set is expressive enough to derive other operators over collections as well as LINQ-style query expressions. The goal of the work is to study a language whose expressions compile to SQL queries and where types correspond to relational schemas, namely they use refinement types to express SQL table constraints within their type system.

Perhaps the most well-known languages to define business process are Business Process Execution Language (BPEL) and Business Process Modeling Language (BPML), although the former is more widely used than the latter. BPML [9] is a XML-based specification language for expressing executable business processes. Due to its high level abstraction, and even though it is turing-complete, BPML ended up not being adopted by industrial BPM systems like Microsoft's BizTalk and IBM's MQServer, who decided to develop their own language: WSFL and XLANG, respectively. This eventually rendered BPML obsolete and led to the definition of a simpler specification language, BPEL, with joint efforts of Microsoft and IBM (by combining BPML together with WSFL and XLANG).

BPEL [10], also based on XML, is actually used for web services' orchestration. So in BPEL a business process is modelled as the set of actions a participant has in a business interaction (a web service). It is also possible to abstract from business processes themselves by defining business protocols that describe how business processes interact between them, that is, what visible exchange messages they exchange without giving details about their internal behaviour.

There are several industry tools for business process management systems (BPMS) that integrate BPEL with a graphical notation language, the Business Process Model and Notation (BPMN). The goal of these BPMS's tools is to provide an easy, intuitive, and visual tool that enables the description, management and development of business process in a organisation. By using BPMN, a non-expert user can describe easily a business process as a flow chart between the necessary activities for the completion of the business process without having to worry about the technologies behind, how they interconnect, or any other technical detail. That gap is filled by the translation of the diagram into fully executable BPEL processes. Some BPMS tools are MQServer, Biztalk, Intalio BPM, ProcessMaker, and Activiti BPM Platform.

6 Concluding Remarks and Ongoing Work

It is our believe that an analysis of the information manipulated in workflows is a fundamental issue that has yet to be addressed by the research community. In order to do so it is fundamental to have abstraction mechanisms for the data itself as well as for the workflows, so a model based on tasks coupled together with data manipulation primitives would be ideal for such analysis.

In this paper, we have proposed a core language to express data-centric processes by presenting a typical DML core augmented with primitives to describe and create tasks/workflows. In our proposal, tasks are first-class citizens which allows us to model richer scenarios such as the project management system (presented in this paper) where we store suspended tasks, when a client makes a submission, in order to be later on resumed by a clerk, when the submission is evaluated.

An interesting direction for this work, which we are at the moment pursuing, consists in investigating how to add information security to the language. Namely, we are interested in information flow analysis to prevent insecure flows of data throughout the execution of a program. For instance, going back to our project management example, we would like to ensure data security concerns such as: (1) only the system's administrator and clerks can fully see entries in the submissions' database; (2) a user can only see project's status, and only of those he submitted; and (3) only the clerk that got assigned to a submission can alter that submission's status as well as execute the stored task for that submission (that has the necessary subtasks to alter the status).

In order to do so, we will apply a type-based approach for information flow analysis. In particular, we will add security labels to values to state which principals or roles can read, write or execute (when concerning a task), forming a lattice of security labels ordered by a partial ordered set $\leq$ such that $\ell \leq \ell'$ if, and only if, ℓ is a more permissive security label than ℓ'. Then we have to ensure by type safety, together with a non-interference property, that our programs do not leak or taint data by preventing flows from higher levels to lower levels. In practice, the non-interference property, means that two instances of the same program, that differ only in their private data (data classified as having higher levels than the output of the program), should have the same output.

Furthermore, in order to express some security policies, we will need to use dependent types. For example, to express security property (3) we would have to add a field, let's say assigned_clerk, in the submission entry to keep track of who is the assigned clerk. Then, we would label the submission field as {Clerk(assigned_clerk)} to state that the principal store in field assigned_clerk that belongs to role Clerk can execute the task stored in field submission. Lastly, we would require some special primitives to determine principals, namely a *this_principal* primitive that would return the user that is executing the code in the current context.

Acknowledgements. We thank the anonymous referees for their insightful comments and suggestions regarding this paper. This work was supported by CITI, INTERFACES Project, and FCT/MCTES under grant SFRH/BD/68801/2010.

References

1. Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. IBM Journal of Research and Development, 1976.
2. Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In Surajit Chaudhuri and Vagelis Hristidis and Neoklis Polyzotis, editor, Proceedings of the ACM SIGMOD International Conference on Management of Data. 2006.
3. Abraham Silberschatz, Henry Korth, and S. Sudarshan. Database Systems Concepts. McGraw-Hill, Inc., publisher, 2006.
4. Microsoft Corporation. The Microsoft code name M Modeling Language Specification Version 0.5, Oct. 2009. Preliminary implementation available as part of the SQL Server Modeling CTP (November 2009).
5. Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David Langworthy. Semantic Subtyping with an SMT Solver, In Proceedings of 15th ACM SIGPLAN International Conference on Functional Programming, Association for Computing Machinery, Inc., September 2010.
6. Gavin M. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Cω. In Proceedings of ECOOP'05, volume 3586 of Lecture Notes in Computer Science, pages 287–311. Springer-Verlag, 2005.
7. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In Proceedinga of FMCO'06, volume 4709 of Lecture Notes in Computer Science, pages 266–296. Springer-Verlag, 2006.
8. Ryan K. L. Ko. A computer scientist's introductory guide to business process management (BPM). In volume 15 of ACM Crossroads. 2009.
9. A. Arkinl Business Process Modeling Language (BPML). Business Process Management Initiative. 2002.
10. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, and S. Thatte. Business Process Execution Language for Web Services, Version 1.1. Specification. BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems. 2003.