Language Design for Validatable Information System Specifications

Daco C. Harkes

Delft University of Technology
`d.c.harkes@tudelft.nl`

1 Introduction

A programming language design must strike a balance between *validatability*, *expressiveness*, and *efficiency*. Bridging the gap between domain concepts and the encoding of these concepts in a programming language is one of the core challenges of software engineering. The *validatability* of a language is a measure of the size of this gap. In a language with a high validatability index, one can express intent with relatively little encoding, which makes it straightforward to establish that a program 'does the right thing'. Validatability decreases with increasing encoding.

Validatability is often at odds with *expressiveness*, the coverage of a domain of computation by a language. High coverage can typically be achieved by providing low level operations that can be combined in many different ways. However, encoding domain concepts in low level primitives increases the distance between intent and realization. Lack of (domain) expressiveness results in an abundance of programming patterns to make up for the missing non-expressible constructs [1]. These programming patterns are an obstacle to understanding of programs by human readers [1], and thus reduce validatability.

Validatability is also at odds with *efficiency*. Realizing a high performance implementation typically requires invasive changes to a basic expression of intent. A language that supports separation of concerns between expression of intent and expression of performance requirements avoids such invasive changes. In general purpose languages efficient computations can be encoded, but at the loss of validatability. In domain-specific languages, such as IncQuery [12] for graph pattern matching, efficiency can be realized with limited encoding, but at the expense of limiting expressiveness to a narrow domain.

The objective of our research group is to investigate the balance between validatability, efficiency, and expressiveness in the design of (domain-specific) programming languages in order to reduce the complexity of software systems.

The objective of my thesis work is to do a case study to investigate this balance in the domain of *information systems*. Information systems are systems for the collection, organization, storage, and communication of information. Information systems aim to support operations, management and decision-making. In order to do this, the data in information systems is filtered and processed to create new data.

Information systems are an interesting domain because both validatability and efficiency are important information system properties. Validatability for information systems is useful because its end users should be able to validate that their system does the right thing. End users should be able to inspect information system specifications, just like they are able to inspect the formulas in a spreadsheet, to validate them. Efficiency for information systems is important because the amount of information and the amount of users tends to grow over time, and the filtering and processing to create new data can depend on a lot of data.

So what is needed for such an information systems language? First, that this language can specify the structure of the information to be collected, organized, stored, and communicated: a *data model*. Second, that this language can specify filtering and processing of data to create

```
entity Submission {
  answer     : String?
  grade      : Float?  = if(childPass) childGrade else no value (default)
  pass       : Boolean = grade >= 5.5 <+ false
  childGrade : Float?  = avg(children.grade)
  childPass  : Boolean = conj(children.pass)
}
relation Submission.parent ? <-> * Submission.children
```

Figure 1 An aspect of a learning management system in which students solve assignments expressed in IceDust. The student `Submission`s form a tree where leafs represent single questions and non-leaves represent labs, or even a full course. Leaf submissions are graded by assigning a grade to the `grade` attribute, while the grades of non-leaf submissions depend (indirectly) on the grades of their child submissions. Note that students only receive a grade for a non-leaf submission if all of its children `pass`, and a submission only `pass`es when its `grade` is sufficient.

new data: the specification of *derived data* (or *derived values* for individual values). This language should enable specification of data models and derived values with direct expression of intent, so that the specifications are validatable. Moreover, the implementations generated from these specifications should be efficient, and the language itself should be expressive enough to enable specification of a wide range of data models and derived values.

2 Related Work

A variety of languages and libraries can be used to specify data models and derived values. However, all of these have their limitations when used to specify information systems. Either the specifications lack validatability, or the implementations lack efficiency, or the languages themselves lack expressiveness. In this section we will describe the general limitations of the object-oriented, functional reactive programming (FRP), and relational paradigms. In general these paradigms are incomparable, but restricted to the domains of data models and derived values they can be compared. As running example we use (an aspect of) a learning management system (Figure 1).

Object-oriented languages and FRP libraries suffer from the lack of bidirectional associations. Data models often contain bidirectional associations, but references in object-oriented languages are unidirectional, enforcing an encoding for bidirectional associations [6]. On the other hand, relational databases enable bidirectional relations through foreign keys, but navigating these relations is through verbose queries, which hampers validatability [6].

Derived value computations often include optional or multiple values. Object-oriented languages and FRP libraries encode the cardinality of values through lists and `null` (or `None`) values. They encode the operations on these cardinalities of values with `maps`, `flatMaps` and `null`-checks (Figure 3). Conversely, relational databases always work with multiple rows. However, columns can still contain `null` values, and view definitions might not be complete. For example, in SQL one must deal with `null` values and missing rows by means of `null`-checks and `unions` (Figure 2).

Object-oriented programming languages can express derived values through getters containing code that calculates a derived value. These getters are a direct expression of intent but do not provide efficiency as the derived value is recalculated each time it is read. Efficiency can be gained by encoding patterns such as caches, but this reduces validatability.

Functional reactive programming aims to limit the encoding patterns for caching and incremental cache maintenance by providing macros (or functions) that encapsulate caching behavior [8, 11]. However, the use of these macros still is an encoding pattern (see Figure 3).

```
CREATE TABLE submission2(
  id int(10) NOT NULL AUTO_INCREMENT, PRIMARY KEY (id),
  parent int(10), answer varchar(20), manualGrade float(4,2));

CREATE VIEW submission2childgradeChildpass AS
SELECT parent AS id, AVG(grade) AS childGrade, BIT_AND(pass) AS childPass
FROM submission3full GROUP BY parent
UNION SELECT id, 0.0 AS childGrade, TRUE AS childPass
FROM submission2 WHERE NOT EXISTS(
  SELECT NULL FROM submission3full WHERE submission3full.parent = submission2.id);

CREATE VIEW submission2grade AS
SELECT id, IF((manualGrade IS NOT NULL), manualGrade,
       IF(childPass, childGrade, NULL)) AS grade
FROM submission2 NATURAL JOIN submission2childgradeChildpass;

CREATE VIEW submission2pass AS
SELECT id, IF((grade IS NOT NULL), (grade >= 5.5), 0) AS pass
FROM submission2grade;

CREATE VIEW submission2full AS
SELECT id, parent, answer, manualGrade, childGrade, childPass, grade, pass
FROM submission2 NATURAL JOIN submission2childgradeChildpass
NATURAL JOIN submission2grade NATURAL JOIN submission2pass;
```

Figure 2 Learning management system grade calculation with MySQL views. The definition is non-recursive as recursive views are not supported in MySQL. Derived values that depend on each other require separate views. As `join`s can omit rows `union`s with default values are required.

```
class Submission {
  val children    : VarSynt[List[Submission]] = Var(Nil)
  val answer      : VarSynt[String]            = Var("")
  val manualGrade: VarSynt[Option[Double]]    = Var(None)
  val childGrade: DependentSignal[Option[Double]] = Signal {
    val grades = children().flatMap( _.grade() )
    if (grades.nonEmpty) Some(grades.sum / grades.length) else None
  }
  val childPass: DependentSignal[Boolean] = Signal {
    children().map( _.pass() ).conjunction
  }
  val grade: DependentSignal[Option[Double]] = Signal {
    manualGrade() match {
      case Some(g) => Some(g)
      case None    => if (childPass()) childGrade() else None
    }
  }
  val pass: DependentSignal[Boolean] = Signal {
    grade().exists( _ >= 5.5 )
  }
}
```

Figure 3 Learning management system with grade calculation with REScala signal macros. The order of the `val`s is declare before read, except for indirect reads (such as the grades of children).

This encoding pattern also changes the types, and the interface. Reading from a plain Scala value is `foo`, while in Scala.React and REScala this is `foo.getValue` and `foo.get` respectively. Writing to a plain Scala variable is `foo = x`, while in Scala.React and REScala this is `foo() = x`. This makes switching calculation strategies hard, the efficiency concern and the derived value specification are not properly separated.

By contrast, relational engines can allow easy switching between calculate-on-read and calculate-on-write as they support both materialized and non-materialized views for calculating derived values [3, 4]. Other relational engines (such as i3QL [10] and IncQuery [12]) only provide materialized views, with the rationale that in every intended load scenario this will be faster. However, relational engines only provide limited expressiveness for recursion (in order to guarantee termination). Moreover, specifying derived values through views requires encoding, especially when the derived values depend on each other, as they need to be in separate view definitions (Figure 2).

3 Data Modeling and Derived Value Computation in IceDust

The objective of my thesis work is to do a case study to investigate the balance validatability, efficiency, and expressiveness in the domain of information systems: IceDust. IceDust eases validation of information system specifications by enabling direct expression of intent for data models and derived values through *bidirectional relations* and *native multiplicities* [7]. In IceDust the efficiency concern is separated from the derived value specification by providing different *calculation strategies* as compiler options. IceDust provides efficiency in a variety of load scenarios through three different calculation strategies.

Bidirectional relations In Object-oriented languages bidirectional associations have to be encoded in multiple unidirectional references (which have to be kept consistent). SQL supports bidirectional relations by foreign keys, but encodes navigation with verbose joins. To avoid encoding, IceDust supports *bidirectional relations*:

```
relation Submission.parent ? <-> * Submission.children
```

Concise navigation is supported with member-access navigation in both directions:

```
mySubmission.children
mySubmission.parent
```

Native Multiplicities Object-oriented languages encode the operations on optional and multiple of values with `maps` and `flatMaps` and `null`-(or `None`)checks (Figure 3). In relational languages such as MySQL these are encoded with `null`-checks and `unions` (Figure 2). To avoid these encodings, IceDust adopts *native multiplicities* [6]. IceDust lifts all its operators, avoiding encodings. Accessing an attribute, for example, is simply a projection:

```
children.grade
```

The type system knows how many values an expression returns (multiplicity denoted by `~`, where `*` is [0,n), `+` is [1,n), `?` is [0,1], and `1` is [1,1]):

```
mySubmission                      // : Submission ~ 1
mySubmission.children             // : Submission ~ *
mySubmission.children.grade       // : Float      ~ *
avg(mySubmission.children.grade)  // : Float      ~ ?
```

Derived Values *Derived value attributes* provide a calculation strategy agnostic way of specifying derived values:

```
entity Submission {  childGrade : Float? = avg(children.grade)  }
```

The calculate-on-read strategy provides performance in load scenarios with many writes and little reads. The calculate-on-write strategy provides performance in load scenarios with many reads and little writes. The calculate-eventually strategy provides maximum performance in many load scenarios but does not provide consistency. All of these strategies support concurrent interactions (through a relational database), which is desirable in information system load scenarios.

4 Future Work

While IceDust eases validation of information system specifications and provides efficiency in a variety of load scenarios, it can be improved. IceDust, in its current form, lacks expressiveness (for example relational joins), and could be efficient in more load scenarios by incorporating more calculation strategies and the composition of calculation strategies.

We want to do a study (survey paper) comparing incremental computation languages and libraries. This study should start with a set of information system use cases for which validatable specification and efficient implementation is desired. Second, it should study a set of state of the art incremental computation tools (languages, engines, and frameworks). For each tool it should be assessed what information systems can be expressed in it, and what mechanics it employs for incremental computation. We hope that the insights from this study will guide what language constructs to add to IceDust to increase its expressiveness.

Next, we would like to extend IceDust in order to increase its expressiveness. This requires adding new mechanics to handle the extra expressiveness efficiently. Adding a relational join operator (and an incremental implementation) is a possibility, as this would allow for derived relations (as opposed to only derived attribute values).

We would also like to extend IceDust with composition of calculation strategies. Composition of calculation strategies increases the number of load scenarios where efficient performance can be provided. Composition of calculation strategies requires changes to the language: it should be specified which derived values should be calculated with which strategy. A static analysis should only admit sound compositions. Experiments should verify that the composition of calculation strategies indeed improves efficiency in certain load scenarios. These benchmarks should illustrate performance trends and trade-offs when switching calculation strategies. It is not our goal to be faster than any competitor for any specific calculation strategy in any specific scenario, rather our goal is to provide easy switching of strategies to provide relatively fast performance in many scenarios.

5 Validation Strategies

We would like to validate IceDust (as in research validation) by applying it in a variety of information systems. The information systems that we would like specify in IceDust are: a learning management system with grade calculation (as mentioned in this paper), statistics from edit scenarios, and duration predictions for student assignments; a finance system with transactions and pivot tables; and a scientific publications system with citation graphs with non-trivial data such as author aliasing. The load scenarios that we are going to use for efficiency validation are going to be recorded from the real systems in use. This should validate the validatability of the specifications, the efficiency of the implementations, and IceDust's expressiveness.

We would also like to validate IceDust by comparing it with other state of the art systems. The state of the art systems we would like to compare with are LogiQL [2], Nominal

Adapton [5], Scala.React [8], Reactive Extensions [9], i3QL [10], REScala [11], and IncQuery [12]. Validatability, efficiency, and expressiveness in the domain of data models and derived value calculations can all be compared. Validatability can be measured by the number of concepts a programmer has to understand in order to understand the code. Less concepts to understand means better validatability (analogous with encodings [1]). To measure the number of concepts a programmer has to understand we plan to digest the hard parts of the information systems mentioned above (in the same way the running example of this paper is the hard part of a larger system), and express these tiny, hard parts in all the state-of-the-art systems. Efficiency can be compared by running benchmarks. We will measure throughput (requests per second) and latency under peak load. The load scenarios for these benchmarks will be digested from the recorded load scenarios of the real systems. Expressiveness can be compared by articulating use cases which can be expressed in one system, but not in another.

With both validation strategies the threat to validity is that information systems and their load scenarios are not representative for other information systems and their load scenarios. We try to limit this threat by covering a variety of systems and a variety of load scenarios taken from real world applications.

Acknowledgements This research was funded by the NWO VICI *Language Designer's Workbench* project (639.023.206).

References

1 Matthias Felleisen. On the expressive power of programming languages. In *ESOP'90*, pages 134–151. Springer, 1990. doi:10.1016/0167-6423(91)90036-W.

2 Todd J. Green. Logiql: A declarative language for enterprise applications. In *PODS*, pages 59–64, 2015. doi:10.1145/2745754.2745780.

3 Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *FTDB*, 5(2):105–195, 2013. doi:10.1561/1900000017.

4 Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *DEBU*, 18(2):3–18, 1995.

5 Matthew A Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *OOPSLA*, pages 748–766, 2015. doi:10.1145/2858965.2814305.

6 Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In *SLE*, pages 241–260, 2014. doi:10.1007/978-3-319-11245-9_14.

7 Daco C. Harkes, Danny M. Groenewegen, and Eelco Visser. Icedust: Incremental and eventual computation of derived values in persistent object graphs. In *ECOOP*, 2016.

8 Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*, pages 707–731, 2013. doi:10.1007/978-3-642-39038-8_29.

9 Erik Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *CUFP*, page 11, 2010. doi:10.1145/1900160.1900173.

10 Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3ql: language-integrated live data views. In *OOPSLA*, pages 417–432, 2014. doi:10.1145/2660193.2660242.

11 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: bridging between object-oriented and functional style in reactive applications. In *AOSD*, pages 25–36, 2014. doi:10.1145/2577080.2577083.

12 Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *MoDELS*, pages 653–669, 2014. doi:10.1007/978-3-319-11653-2_40.