The TTC 2014 Movie Database Case: Rascal Solution[*]

Pablo Inostroza
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
storm@cwi.nl

Rascal is meta programming language for processing source code in the broad sense (models, documents, formats, languages, etc.). In this short note we discuss the implementation of the "TTC'14 Movie Database Case" in Rascal. In particular we will highlight the challenges and benefits of using a functional programming language for transforming graph-based models.

Introduction

Rascal is a meta programming language for source code analysis and transformation [1, 2]. Concretely, it is targeted at analyzing and processing any kind of "source code in the broad sense"; this includes importing, analyzing, transforming, visualizing and generating, models, data files, program code, documentation etc.

Although at first sight Rascal may seem an imperative language (e.g., due to the Java-like syntax), it is a functional programming in the sense that all data is immutable (implemented using persistent data structures). Functional programming concepts are at the core of the language: algebraic data types, pattern matching, higher-order functions, comprehensions etc.

Specifically for the domain of source code manipulation, however, Rascal features powerful primitives for parsing (context-free grammars), traversal (visit statement), relational analysis (transitive closure, image etc.), and code generation (string templates). The standard library includes programming language grammars (e.g., Java), IDE integration with Eclipse, numerous importers (e.g. XML, CSV, YAML, JSON etc.) and a rich visualization framework.

Rascal is not a model-transformation system in the narrow sense of the term. In the context of model transformation we identify the following challenges:

- Since Rascal is based on immutable data, models have to be represented as (containment) trees with explicit, super-imposed cross-references. As a result, some transformation have to explicitly look up model elements, given their identity.
- For the same reason, model transformation amounts non-destructive rewriting. That is, it is impossible to perform in-place updates. This has benefits for reasoning (locality), but might affect performance.

In the following sections we discuss the realization of the TTC'14 tasks in Rascal. We conclude the paper with some observations and concluding remarks. All code examples can be found online at:

`https://github.com/cwi-swat/ttc2014-movie-database-case`

[*]This research was supported by the Netherlands Organisation for Scientific Research (NWO) Jacquard Grant "Next Generation Auditing: Data-Assurance as a service" (638.001.214).

Submitted to:
TTC 2014

Representing IMDB in Rascal

Since Rascal is a functional programming language, where all data is immutable, the IMDB models have to be represented as trees instead of graphs. If there are cross references in the model, these have to be represented using (symbolic or opaque) identifiers which can be used to look up referenced elements. We use an algebraic data type to model IMDB models.

The following data type captures the structure of IMDB models:

```
data IMDB   = imdb(map[Id, Movie] movies, map[Id, Person] persons,
                   set[Group] groups, rel[Id movie, Id person] stars);
data Movie  = movie(str title, real rating, int year);
data Person = actor(str name) | actress(str name);
data Group  = couple(real avgRating, Id p1, Id p2, set[Id] movies);
```

An IMDB model is constructed using the `imdb` constructor. It contains the set of movies, persons, groups and a relation `stars` encoding which movie stars which persons. Both movies and persons are identified using the opaque `Id` type. To model this identification, the `movies` and `persons` field of an IMDB model are maps from such identifiers to the actual movie resp. person.

Movies and persons are simple values containing the various fields that pertain to them. The `Group` type captures couples as required in Task 2. A `couple` references two persons and a set of movies using the opaque identifiers `Id`.

Task 1: Synthesizing Test Data

Synthesizing test data creates values of the type `IMDB` as declared in the previous section. The process starts with an empty model (`imdb((), (), {}, {})`[1]), and then consecutively merges it with test models for a value in the range $1,...,n$. Each test model in turn consists of merging the negative and positive test model as discussed in the assignment. As an example, below we list the creation of the positive test model:

```
IMDB createPositive(int i) = imdb(movies, people, {}, stars)
    when movies := ( j:  movie("m<j>", toReal(j), 2013) | j <- [10*i..10*i+5] ),
         people := ( 10*i:  actor("a<10*i>"), 10*i+1:  actor("a<10*i+1>"),
                     10*i+2:  actor("a<10*i+2>"), 10*i+3:  actress("a<10*i+3>"),
                     10*i+4:  actress("a<10*i+4>") ),
         stars := {<10*i, 10*i>, <10*i, 10*i+1>, <10*i, 10*i+2>, <10*i, 10*i+3>,
                 <10*i+1, 10*i>, <10*i+1, 10*i+1>, <10*i+1, 10*i+2>, <10*i+1, 10*i+3>,
                 <10*i+2, 10*i+1>, <10*i+2, 10*i+2>, <10*i+2, 10*i+3>,
                 <10*i+3, 10*i+1>, <10*i+3, 10*i+2>, <10*i+3, 10*i+3>, <10*i+3, 10*i+4>,
                 <10*i+4, 10*i+1>, <10*i+4, 10*i+2>, <10*i+4, 10*i+3>, <10*i+4, 10*i+4>};
```

The function uses map comprehensions to create the `movies` and `people` fields, and a binary relation literal to create the `stars` relation. It then simply returns a value containing all those fields.

[1]The syntax `()` indicates an empty map.

Task 2: Adding Couples

Task 2 consists of enriching IMDB models with “couples”: pairs of persons that performed in the same movie, once or more often. This transformation is expressed by updating the `couples` field with the result of the following function:

```
set[Group] makeCouples(IMDB m) {
    costars = toMap(m.stars); couples = ();
    for (mov <- costars, s1 <- costars[mov], s2 <- costars[mov], s1 < s2)
       couples[<s1, s2>]?{} += {mov};
    return { couple(0.0, x, y, ms)
                     | <x, y> <- couples, ms := couples[<x, y>], size(ms) >= 3 };
}
```

This function first convert the binary relation `stars` to a map from movie Id to set of person Ids. The central for loop iterates over all movies and all combinations of two actors and adds the movie to a table maintaining the set of movies for all couples (a map taking tuples of person Ids to sets of movie Ids). The side condition `s1 < s2` ensures we don’t visit duplicate or self combinations. The question mark notation initializes a map entry with a default value, if the entry did not yet exist. In the final statement, a set of Groups is returned containing all couples which performed in 3 or more movies.

Task 3: Computing Average Ratings for Couples

As can be seen in the previous section, the average rating field of couples is initialized to 0.0. In this task we again transform an IMDB model, this time enriching each couple with its average rating of the movies the couple co-starred in.

The following function performs this transformation:

```
IMDB addGroupRatings(IMDB m) = m[groups=gs]
  when gs := { g[avgRating = mean([ m.movies[x].rating | x <- g.movies ])]
                | g <- m.groups };
```

The groups field of the model `m` is updated with a new set of groups, as created in the when-clause of the function. The new set of groups is created using a comprehension, updating the `avgRating` field of each group. The average is computed based on the list of ratings obtained from the movies contained in `m` that are referenced in the group `g`.

Extension Task 1: Top 15 Rankings

The object of the extension Task 1 is to compute top 15 rankings based on the average ratings or number of movies a couple participated in. To represent rankings, we first introduce the following type alias:

```
alias Ranking = lrel[set[Person] persons, real avgRating, int numOfMovies];
```

A ranking is an ordered relation (`lrel`) containing the co-stars, the average rating and the number of movies of a couple. To generate rankings in this type, we create a generic function that takes the number of entries (e.g., 15), an IMDB model, and a predicate function to determine ordering of groups. This last argument allows to abstract over what the ranking is based (e.g., average rating or number of movies).

```
Ranking rank(int n, IMDB m, bool(Group, Group) gt) =
    take(n,
       [<{m.persons[x] | x <- getPersons(g)}, g.avgRating, size(g.movies)>
            | Group g <- sort(m.groups, gt)]);
```

Again, this function employs a comprehension to create a ranking, iterating over the groups in the IMDB model, sorted according to the predicate `gt`. For each person in the group (extracted using `getPersons`), the actual person value is looked up in the model (`m.person[x]`). The `take`, `size` and `sort` functions are in the standard library of Rascal.

The actual top 15 rankings are then obtained as follows:

```
Ranking top15avgRating(IMDB m)    = rank(15, m, greaterThan(getRating));
Ranking top15commonMovies(IMDB m) = rank(15, m, greaterThan(getNumOfMovies));
```

The last argument to rank is constructed using a higher-order function, `greaterThan`, which constructs comparison functions on groups based on the argument getter function (i.e. `getRating` and `getNumOfMovies`). So in the first case, rank is called with a comparison predicate based on average ratings of groups, whereas in the second case, groups are ordered based on the number of shared movies in a group.

Extension Task 2: Generalizing groups to cliques

The extension to task 2 consists of generalizing couples to arbitrarily sized cliques of people who co-starred in the same set of movies. A couple is a special case of this situation, where the clique size is 2. To represent arbitrary cliques in the model, we modularly extended the Group data type (see Section 1.1) as follows:

```
data Group = clique(real avgRating, set[Id] persons, set[Id] movies);
```

This declaration states that the `clique` constructor is now a valid group value, in addition to `couple`.

Enriching an IMDB model with cliques follows the same pattern as enriching a model with couples. In fact the following function follows almost exactly the same structure as the function `makeCouples` described earlier:

```
set[Group] makeCliques(IMDB m, int n) {
    costars = toMap(m.stars); cliques = ();
    for (Id movie <- costars, set[Id] s <- combinations(costars[movie], n))
      cliques[s]?{} += {movie};
    return {clique(0.0, s, ms) | s <- cliques, ms := cliques[s], size(ms) >= 3 };
}
```

Instead of iterating over pairs of actors explicitly, we now iterate over all combinations of size *n* using the helper function `combinations`, which generates all combinations of size *n* taking elements from the set `costars[movie]`.

Extension Task 3 & 4 These are the same as Task 3 and Extension of Task 1 respectively, because the code is polymorph over groups. Only in the latter case, the `getPersons` accessor needs to be extended to accommodate the new `clique` constructor defined in Extension Task 2.

Observations and Concluding Remarks

Looking back at the effort implementing the TTC'14 tasks in Rascal we can observe that Rascal posed no problems for solving the tasks. The solutions are small and declarative. The size of the implementation is around 130 SLOC, including some helper functions, but excluding loading the model from XML which is another 38 SLOC. Although Rascal allows side-effects in (local) variables, with the exception of `makeCouples` and `makeCliques` none of the function use side-effects of any kind.

Another observation is that the tasks mostly involved querying the models and aggregating new result to enrich the model. In such cases, comprehensions are valuable features to create sets, maps, or relations of model elements. Rascal's built-in features for traversal and powerful pattern matching (e.g., deep pattern matching), were not (even) needed to perform most of the tasks in an adequate way.

Related to the nature of the tasks, the fact that cross references had to be represented and managed explicitly posed no problem. In all cases the top-level IMDB model was always available to perform the necessary reference lookups in the `movies` and `persons` tables. It is, however, possible that transformations that actually change the referential structure of a model require more administration to keep referential integrity in tact.

In terms of performance, we observed that the immutability of models was not an inhibiting factor most of the time, but again, this might be influenced by the nature of tasks. In absence of suitable benchmarks to compare, we can at this point only speculate about whether the Rascal implementation is slow. As an example, we observed that extracting cliques > 2 takes much longer than 5 minutes on a 3.3Mb IMDB file. The most probable reason is that the function `combinations` has high computational complexity, even though we use a dynamic programming implementation. One area where destructive, in-place updates could have improved performance is, again, in the couples/cliques code. Because of immutability, we have to generate the full set of all couples/cliques *first*, then filter them, and finally insert them into the model. In an imperative setting, the couples could be inserted directly in the model, after which the model itself could be pruned to remove the couples/cliques with less than 3 movies.

Finally, we would like to point out that Rascal's module system proved its value. Some tasks could be implemented as modular extensions of earlier tasks, combining extension of data types (Extension Task 2) and extension of functions (Extension Tasks 3 and 4).

References

[1] Paul Klint, Tijs van der Storm & Jurgen Vinju (2009): *Rascal: A domain-specific language for source code analysis and manipulation*. In: *SCAM*, pp. 168–177.

[2] Paul Klint, Tijs van der Storm & Jurgen Vinju (2011): *EASY Meta-programming with Rascal*. In João Fernandes, Ralf Lämmel, Joost Visser & João Saraiva, editors: *Generative and Transformational Techniques in Software Engineering III*, *Lecture Notes in Computer Science* 6491, Springer, pp. 222–289.