Recognition Strategy Library (RSL)

Christopher Sasarak (cms5347@rit.edu)
Advisor: Dr. Richard Zanibbi
Computer Science Department, Rochester Institute of Technology

Introduction

Originally imagined as the Recognition Strategy Language, the Recognition Strategy Library is a data provenance tool for Python meant rapidly prototyping new algorithms and storing intermediate states of program data. These data are stored in a tree representing the execution of the algorithm and can be queried or stored to disk for later analysis.
Existing tools VisTrails and Karma Provenance Tool provide data provenance, but neither are program libraries. VisTrails provides a graphical tool which is used to build workflows, or procedures for analysing a data-set[1]. Karma records provenance by managing and recording communications between different web services[3]. In terms of existing systems, RSL is most similar to the query based debugging system described in[2]. Queries about a program are made by specifying a domain of objects to search and a predicate to filter those objects[2]. RSL is simpler than VisTrails and Karma in that it is a library rather than a separate provenance system. This means that a programmer can get started with RSL and prototype algorithms and capture/query provenance data while only needing to learn RSL's API.
Originally, RSLib was the Recognition Strategy Language. RSLang [4] was implemented using Standard ML (SML) as a target language. We had hoped to change this to Python. However, after working with the code-base more my advisor and I decided that we could simplify RSL by making it a library. An advantage of this is that RSL programmers could have access to all of their favorite Python libraries. It also meant that RSL's maintainers would not have to manage parsing and code generation themselves. This poster presents the rethought RSLib.

RSLib Program Elements

There are several basic elements for any RSL strategy.

Strategy A strategy is an object that manages execution in RSL and also stores provenance information.

Interpretation Type An interpretation type is the type of data that the algorithm modeled with RSL will work with. Provided with RSL is a convenient factory function for generating interpretation types.

Report Function These functions are queued to run to do reporting operations at the end of a strategy's run.

Decision Function A decision function is a function which can take an interpretation set and returns a new one. The strategy records information coming from decision functions.

Decision Label These are string labels that are associated with decision functions or reports when queuing. The strategy object also associates them with nodes in the execution trace graph.

Conclusion

While the basic operations of RSL as first envisioned have been implemented. There are still improvements that we would like to make in the future. Basic export of for simple strategies to a sqlite3 database works. For more complex interpretation types like the matrices used in the example, something more complicated code is needed to generate the interpretation tables. One option might be to use an object relational mapper (ORM) like SQLAlchemy to do this.
An additional area of work is testing the library with developers. I implemented the library in a way that I thought was natural, but we can probably find better ways to do things with the input of more developers. One thing that I did notice while working on the skyscrapers puzzle solver was that it started to become inconvenient to wrap and unwrap interpretations and interpretation sets. This ended up making some code difficult to read later on.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. IIS-1016815. This material is alse based upon work supported by the RIT Center for Student Innovation's SURF program. I would also like to thank Dr. Richard Zanibbi for his guidance in completing this project.

Example

This example is meant to show off how to create and use an RSL strategy with RSL. This program is a solver for a puzzle called Skyscrapers. Skyscrapers is a grid puzzle, an example board is shown in Figure 1.

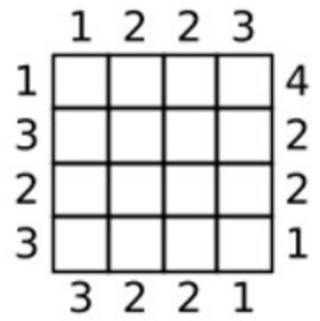

Figure 1: Board

The idea is to fill the board with numbers which satisfy the constraints around the grid.

1. The numbers outside the grid say how many skyscrapers a person can see from that position.
2. Each row and column can only have one skyscraper of a given height.

I implement the solver as a backtracking algorithm. In backtracking, we recursively generate successor states to starting from an initial state. In this case our initial state is an empty board, and the solution is a full board which satisfies the constraints. RSL will help capture the intermediates board states as they are generated by my algorithm. A snippet of the code I use to set up and run the strategy follows.

```
interp_t = rsl.makeInterpretationType("state", "mat")

strat = rsl.Strategy(interp_t, [interp_t(initial_state)])

# Generate a trampoline function to run our algorithm
solver = solve(constraints)
trampolined = strat.mk_trampoline(solver)

strat.append((trampolined, "solve_puzzle"))

print "\nGet solution from a zeroed matrix."
reports = strat.run()

leaves = strat.collectInterps(".+leaf.+")
solutions = strat.collectInterps("solve_puzzle")
```

In order to store the intermediate states of this recursive algorithm, I use a strategy called "trampolining". Instead of a recursive call at the end of the solver function, I return the intermediate states with the next function to call. The Strategy records interpretations and then calls the next step of the function. I then append the `trampolined` function to the list of decision functions with the decision label `solve_puzzle`.
After running the strategy, the `strat` object contains all the interpretations used by the algorithm. Since trampoline deals with many intermediate steps, it automatically labels them using an id number. In the case where a non-solution leaf is found in the recursion tree, it is labeled as such. Using `collectInterps` I query for interpretations that were a full board, but not a solution. Likewise I query for for the solution node with the decision label "solve_puzzle". A snippet of the output from this last query follows.

```
...
solutions:
{m = [[4 3 2 1]
 [1 2 4 3]
 [3 4 1 2]
 [2 1 3 4]]}
```

References

[1] Juliana Freire, ClÃąudio T. Silva, Steven P. Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. Managing rapidly-evolving scientific workflows. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, number 4145 in Lecture Notes in Computer Science, pages 10–18. Springer Berlin Heidelberg, January 2006.

[2] Raimondas Lencevicius, Urs HÃűlzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 304âĂŞ317, New York, NY, USA, 1997. ACM.

[3] Y.L. Simmhan, B. Plale, and D. Gannon. A framework for collecting provenance in data-centric scientific workflows. In *International Conference on Web Services, 2006. ICWS '06*, pages 427–436, September 2006.

[4] Richard Zanibbi. Programmer's guide to the recognition strategy language (RSL), 2011.