Linear Temporal programming in an Esoteric Programming Language

Contact: [email]@grinnell.edu

Tristan Knoth
Grinnell College
[knothtri17]

Reilly Grant
Grinnell College
[grantrei]

Chris Kottke
Grinnell College
[kottkech17]

Abstract

For this project we will develop an esoteric programming language based on the popular cartoon Rick and Morty. This programing language will support multithreading and will also make use of integrated Linear Temporal logic to verify the correctness of multithreaded programs. In addition, this language is intended to be accessible enough to attract students and programmers unfamiliar with the concepts of temporal logic and parallelism.

1. Introduction

The popular TV show Rick and Morty uses the concepts of parallel timelines and alternate universes as a main theme in many episodes. Given the popularity of the show and its relevance in pop culture, we decided that it would be an appropriate medium through which to introduce the concepts of multithreading and temporal logic. The language's syntax will include many references to the show, and thus be attractive to fans of the show. It will also make the use of temporal logic more whimsical and thus increase the topic's appeal to a diverse audience.

Our final goal is to create a Turing complete language which satisfies the above conditions of reference, and also makes serious use of linear temporal programming and its related concepts.

2. Prior Work

Linear Temoral Logic is a mathematical framework that deals with statements whose truth value can change over time. It has been used in programming for software verification, specificly in when working with concurrent programs[1, 6]. Temporal Logic has also been used in declarative programming languages to increase the expressiveness[5]. We could not find a non declarative programming language that supports temporal logic.

Additional work that will be important to this project comes from the field of language design. Language design is a field which has had a lot of focus directed towards it. There are currently hundreds of languages that have been created purely to explore the bounds of programming languages, and the field of programming language research has been growing rapidly in recent years [10]. It is easy to find advice on creating programming languages, such as that given by Dominic Orchard of the university of Cambridge [8]. With the large amount of work that has been done on programming languages, we predict that finding resources to assist us will not be extraordinarily difficult.

3. Proposed Work

Temporal logic can be an extremely useful concept in various computing problems, particularly related to program verification. In order to ensure, for example, that a certain state is eventually reached, or that some data will eventually have some type, we can use temporal logic. We similarly, we can use checking with temporal logic to ensure that data that should not be accessed again, such as pointers, are never used again after a certain point. A final important application we can use temporal logic for is to ensure "fairness" in multi-threaded systems. A program may, for example, want to ensure that if a thread wants to use a certain data that request is eventually fulfilled, while also ensuring that other threads won't have the data that they are using suddenly change.

Thus, we propose to use Haskell to develop a simple Turing-complete computer language featuring temporal logic and some simple multi-threading utilities that are also simple to the user. Because of the advantages that using temporal logic gives to concurrent operations, we cannot use the temporal logic capabilities to its full potential without this multithreading. GHC already supports implicit parallelism. Thus, we can simply use GHC's implementation as the foundation of our multi-threading capabilities. We also intend for our language to support adding various temporal logic statements regarding values to the program. This will allow programmers to properly verify non-terminating applications.

Due to the scope of this challenge, and the variety of choices that are possible in language design, we are not entirely sure of the final design of our languages. We will continually be in communication with our professor while making the decisions about what language features to focus on in our limited time.

4. Timeline

1. Friday 11/4: project checkpoint 1 due:

   By Friday 11/4, we intend to have a basic Turing complete language with many of the jokes and references of Rick and Morty Implemented.

2. Friday 11/18: project checkpoint 2 due:

   By Friday 11/18 we intend to have implemented singly threaded temporally logical system integrated into the language. This will be reminiscent of Haskell's lazy execution due to it's singly threaded nature.

3. Friday 12/2: project checkpoint 3 due:

   By Friday 12/2 we intend to have easily implemented a system which allows the user to simply handle multithreading, as well as serial execution.

4. Monday 12/5 and Wednesday 12/7: project presentations:

   By this point, we intend to have finished the project, and mostly be cleaning it presenting to our peers.

5. Friday 12/9: final project deliverables due:

By this point, we will have the basic all implemented, or at least some versions of all of them. We may have remaining features that we were unable to implement due to the scope of the project, but complete basics implementation will be done.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL `http://doi.acm.org/10.1145/5397.5399`.

[2] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 736–747, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884798. URL `http://doi.acm.org/10.1145/2884781.2884798`.

[3] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):6195, 1991. doi: 10.1016/0004-3702(91)90006-6.

[4] J. Duregård and P. Jansson. Embedded parser generators. *SIGPLAN Not.*, 46(12):107–117, Sept. 2011. ISSN 0362-1340. doi: 10.1145/2096148.2034689. URL `http://doi.acm.org/10.1145/2096148.2034689`.

[5] J. Gaintzarain and P. Lucio. Logical foundations for more expressive declarative temporal logic programming languages. *ACM Transactions on Computational Logic*, 14(4):141, Jan 2013. doi: 10.1145/2528931.

[6] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. *Protocol Specification, Testing and Verification XV IFIP Advances in Information and Communication Technology*, page 318, 1996. doi: 10.1007/978-0-387-34892-6_1.

[7] P. Jonsson and C. Bckstrm. A unifying approach to temporal constraint reasoning. *Artificial Intelligence*, 102(1):143155, 1998. doi: 10.1016/s0004-3702(98)00031-9.

[8] D. Orchard. The four rs of programming language design. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 157–162, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0941-7. doi: 10.1145/2089131.2089138. URL `http://doi.acm.org/10.1145/2089131.2089138`.

[9] D. Orchard. The four rs of programming language design. *Proceedings of the 10th SIGPLAN symposium on New ideas,new paradigms, and reflections on programming and software - ONWARD '11*, 2011. doi: 10.1145/2089131.2089138.

[10] z. Language list. URL `https://esolangs.org/wiki/language_list`.