Mukul Rathi
Queens'
msr45

Part II Computer Science Project Proposal

Types for Data-Race Freedom

14 October 2019

Project Originator: Mukul Rathi and Alan Mycroft

Project Supervisor: Alan Mycroft

Director of Studies: Alastair Beresford

Overseers: Jon Crowcroft and Thomas Sauerwald

Introduction and Description of the Work

Data races occur when two threads are trying to access the same data address concurrently, leading to inconsistent state. For example, a data race may occur when trying to credit a bank account whilst calculating interest on the current amount. Data races are notoriously difficult to detect and reproduce, since it is difficult to predict the order of execution of large multi-threaded systems, and some orders of execution are so seldom traversed that bugs only manifest years later. These bugs are so prevalent in modern concurrent programs that they've been dubbed *Heisenbugs*.

Traditional mechanisms of preventing data races such as locking place the onus on programmers to correctly enforce the guarantees, which leads to other issues like deadlock.

The solution therefore is for the type system to statically guarantee data-race freedom, a notable example being Rust's type system's *ownership* feature - which only allows one thread to own a reference to the data at any given time. However Rust's *ownership* feature does not support the Multi-Reader Single Writer concurrency design pattern, where multiple threads can read the data concurrently (since reading doesn't alter the data). The concurrent object-oriented programming languages Encore and Pony have a richer type system, which associates a capability with each reference based on the type of data access (read/write) and number of threads accessing the reference. This affords a greater degree of concurrency whilst still providing the same static guarantees.

In this project, I will create an interpreter for a simplified object calculus henceforth provisionally dubbed *Bolt* and adapt the static type system Kappa [1] used in Encore to implement a static type checker *Pauli* for the calculus. I will demonstrate that *Pauli* enforces the property of data-race freedom in *Bolt* and thus excludes this class of *Heisenbugs* from the programs that type-check.

Starting Point

I have no previous experience in implementing type checkers or interpreters, however I studied Semantics of Programming Languages and Compiler Construction. The Concurrent and Distributed Systems course I studied last year will be relevant when identifying programs that have data races.

Success Criteria

For the project to be deemed a success, the following items must be successfully completed:

1. Design the simplified object calculus *Bolt* in which data races can occur. This should

support classes with directly-accessible fields (no methods), first-order functions, and structured concurrency in the form of explicit forking and synchronisation points.

2. Implement an interpreter for *Bolt*. This will use a random scheduling algorithm to interleave the thread computations in order to simulate potential data races.

3. Implement the type checker *Pauli* for the calculus *Bolt*. This should distinguish between data references that are exclusive to a thread and those that are shared.

4. Create a corpus of *Bolt* programs that can be used to evaluate the static type checker *Pauli*, with a subset exhibiting potential data race conditions.

Evaluation

The structure of the project does not lend itself to quantitative evaluation, so I will instead evaluate *Pauli* using the test suite of *Bolt* programs created as part of the success criteria.

The corpus of *Bolt* programs will be handcrafted, taking inspiration from the Concurrent and Distributed Systems course, which gave pseudocode of programs with concurrency bugs. In addition, I will look at simple concurrent programs from other languages such as Java, and map these to the *Bolt* calculus.

I will evaluate the program's output when run using the interpreter (noting any data races that may have occurred) and compare this with whether the program type checks using *Pauli*.

In addition to checking the soundness of *Pauli* using the test suite (programs with data races do not type check), I will also evaluate the expressiveness of *Pauli* by analysing which data-race-free programs do not type check.

Possible Extensions

The following is a list of possible extensions to the project:

1. Augment interpreter to maintain state about each thread's data accesses to determine if *any* concurrent execution order has a data race.

2. Add ability to compose reference type capabilities in *Pauli*, and introduce a hierarchy of reference types for nested data references.

3. Implement data-race-free data structures using *Pauli*, such as Binary Search Trees.

4. Extend *Pauli* based on follow-up papers related to Encore - again evaluating with a corpus of *Bolt* programs to determine the increase in expressiveness.

5. Add type inference to *Pauli*.

Timetable and Milestones

Weeks 1 to 2 (25 Oct 19 - 7 Nov 19)

Proposal Submitted

Read up on OCaml syntax and the resources available (e.g. lexers and parsers) for writing an interpreter and type-checker in OCaml.

Read ahead in the Optimising Compilers course about liveness analysis in preparation for type-checking the references.

Weeks 3 to 4 (8 Nov 19 - 21 Nov 19)

Design the language syntax for *Bolt*.

Implement the lexer and parser for the language syntax designed in the previous block, and write tests to verify the output abstract syntax trees are correct.

Milestone: Given *Bolt* programs, generate the abstract syntax trees.

Weeks 5 to 6 (22 Nov 19 - 5 Dec 19)

Implement the interpreter for *Bolt*. Write some *Bolt* programs and verify the interpreter executes them correctly.

Milestone: Run *Bolt* programs using the interpreter.

Weeks 7 to 9 (6 Dec 19 - 26 Dec 19)

End of Michaelmas Term - start of Christmas holidays.

Implement *Pauli* and continue to write further test *Bolt* programs and verify *Pauli* type checks them correctly.

Take time off for Christmas.

Milestone: Given a *Bolt* program, determine whether it type-checks using *Pauli*.

Weeks 10 to 12 (27 Dec 19 - 9 Jan 20)

Slack time to finish off any implementation.

Milestone: Finish core project implementation.

Weeks 13 to 14 (10 Jan 20 - 23 Jan 20)

End of Christmas holidays - start of Lent Term.

Prepare further test cases for the core project evaluation. Draft the Progress Report and discuss this with supervisor well ahead of deadline.

Milestone: Finish core project evaluation.

Weeks 15 to 16 (24 Jan 20 - 6 Feb 20)

Finalise progress report and presentation.

Start work on some of the possible project extensions if time allows - adjust timetable based on current progress. Primarily focus on Extension (2) - adding the ability to compose reference type capabilities.

Deadline (31 Jan): Submit progress report.

Weeks 17 to 18 (7 Feb 20 - 20 Feb 20)

Continue to work on Extension (2) and implement more complex test *Bolt* programs that use these new composed reference capabilities.

Start writing drafts for Introduction and Preparation chapter.

Milestone: Finish implementation of Extension (2).

Milestone: Complete Draft Introduction and Preparation chapters.

Weeks 19 to 20 (21 Feb 20 - 5 Mar 20)

If time permits, implement Extension (3) - i.e. create a set of *Bolt* data-race-free data structures using the primitives in the language.

Wrap up work on extensions and focus on writing draft Implementation chapter.

Milestone: Implement data-race-free data structures in *Bolt*.

Milestone: Complete Draft Implementation chapter.

Weeks 21 to 23 (6 Mar 20 - 26 Mar 20)

End of Lent Term - start of Easter holidays.

Write up draft Evaluation chapter, and discuss evaluation with supervisor.

Based on feedback, write up more tests and evaluation metrics if needs be (slack time to continue extended evaluation of project).

Finish Conclusions chapter. Aim to send first draft to supervisor for feedback.

Milestone: Finish first draft of dissertation.

Weeks 24 to 25 (27 Mar 20 - 9 Apr 20)

Revise for exams whilst waiting on supervisor feedback.

Week 26 (10 Apr 20 - 16 Apr 20)

Incorporate feedback from supervisor and submit draft to Director of Studies. Spend time going through code repository and checking code style and layout.

Week 27-30 (17 Apr 20 - 1 May 20)

Revise for exams. Slack time in case additional evaluation and tests need to be written. **Milestone (1 May):** Submit Dissertation!

Resource Declaration

I will be using my personal laptop (MacBook Pro 2016 - 2.6GHz i7, 16GB RAM) as my primary machine for software development, and will use the Computing Service's MCS as backup, along with periodic backups to Google Drive and Git version control.

References

[1] Elias Castegren and Tobias Wrigstad. Reference capabilities for trait based reuse and concurrency control, 2016.