Formalizing a relational model of concurrent programs in a dependently typed environment[4]

István Donkó[†], **Ambrus Kaposi**[‡], **Melinda Tóth**[§]

ELTE, Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
[†]`isti115@inf.elte.hu` [‡]`akaposi@inf.elte.hu` [§]`tothmelinda@elte.hu`

Sequential programming languages have already been formalized in dependently typed programming languages, such as for example Agda or Coq, but the computer based formalization and verification of concurrent programs is still in its early days. The goal of our research is to formalize a relational model [4] that describes the behavior of distributed concurrent programs in a theorem prover system. Our long term goals include the formalization of the material of the subject titled *"Specification and Implementation of Distributed Systems"*, which serves as a core part in the Computer Science education at Eötvös Loránd University in Budapest.

Since our goal is not the introduction of a new way to address the problem of creating correctness proofs for parallel programs, rather the adaptation of an existing system for formalized implementation, instead of discussing the theoretical methods, here we focus on a more practical approach and explore what others have achieved in the field of computer based verification.

Motivation Software plays a critically important role in the life of modern societies. More or less everybody interacts with computer programs on countless occasions during our everyday lives, most of the time probably not even noticing. For example, just paying with a credit card while shopping, being able to call someone with our mobile phones, or even as mundane tasks, as operating modern home appliances, like washing machines or microwave ovens requires interaction with software. Problems occurring in these example situations mostly just cause inconveniences, but if we take the more critical scenarios into account, such as for example software running on an airplane, or keeping a nuclear power plant safe, we can see that programmers have an even bigger impact.

Borrowing from Robert C. Martin we can define the beginning of programming around the work of Alan Turing, since he was the first one, who wrote code for machines in the sense that we would recognize today. In his time, he described the future possibilities of his vision with the following sentences in a lecture to the London Mathematical Society [6].

"In order to supply the machine with these problems we shall need a great number of mathematicians of ability. These mathematicians will be needed in order to do the preliminary research on the problems, putting them into a form for computation."

He stated the need for mathematicians for the precise formalization of problems. As we can see today, his approach is necessary for building critical systems, the correctness of which can decide between life and death, we need to have formal strategies to verify behaviors of programs under all circumstances instead of just observing them for the most likely situations.

Introduction There are lots of existing means for confirming the adherence of simple sequential programs to their specifications, ranging from formal verification procedures carried out on paper to contracts built into programming languages, that can be checked and enforced automatically, either via static code analysis, or during runtime by monitoring different values. Reasoning about

[4]The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16- 2017-00002). The research is part of the "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme no. 2020-4.1.1.-TKP2020 (National Challenges Subprogramme) funding scheme.

parallel programs is a lot more complicated, but this complication also serves as an explanation for the need to do so, since concurrency is often a result of multiple systems working together, in which case it is a lot easier to make mistakes because of the unpredictable order of execution of instructions. Several different ways are known to approach formal proofs of correctness for concurrent programs. For example multiple specific methods can be seen in [2]. What we chose to base our research on is the material of the subject titled *"Specification and Implementation of Distributed Systems"*.

Our formalization does not follow the material exactly, but we tried to stay as close to the original notation as possible, as we also have intentions to later further expand this project to cater for educational usage for example as part of the practical courses. We consider this to be a valuable opportunity for creating a teaching tool that can greatly aid the understanding of the subject for students. The use of proof assistants has already been successfully introduced in several other classes [5], which helps making this idea seem quite feasible.

Background We used type theory (namely, the Agda [1] implementation of it) as the main tool for creating and verifying our formalization, which – by being an expressive alternative foundation for mathematics – enables the formalization of constructive proofs through the connections to intuitionistic logic given by the Brouwer–Heyting–Kolmogorov interpretation. After formalizing a model by defining its types and their elements, one can express statements and theorems in forms of new types, the instances of which can be thought of as proofs for them. This is due to the so called *"propositions-as-types"* paradigm, formally known as the Curry–Howard isomorphism.

Results The main results of our work include the fully formalized version of a big core part from the original model [3] we built our research around, which is precise enough for computer based typechecking. This consists of a definition for a language which contains parallel conditional assignments as well as several types of predicates and statements that can be used to describe specifications for programs written in this language. On top of these foundational constructs we have developed proofs for several generic lemmas and some bigger theorems. We have also formalized a parallelized version of the bubble sort algorithm and the verified some of its properties.

References

[1] Agda Development Team. *Agda - https://github.com/agda/agda*. Version 2.6.1. URL: `https://github.com/agda/agda`.

[2] Robin Candy. "Towards Concurrent Hoare Logic". In: (Nov. 2012). URL: `https://ir.canterbury.ac.nz/handle/10092/14861`.

[3] István Donkó. *Formalizing a relational model of concurrent programs in a dependently typed environment.* Paper at the Student Association Conference, Faculty of Informatics, Eötvös Loránd University, May 2020, Received 1st prize.

[4] Ákos Fóthi and Zoltán Horváth. *Párhuzamos és elosztott programozás (English: Parallel and Distributed Programming).* ELTE, 2005.

[5] Dániel Horpácsi et al. "Interactive Teaching of Programming Language Theory with a Proof Assistant". In: *Central-European Journal of New Technologies in Research, Education and Practice* (2020).

[6] Alan M Turing. "Lecture to the London Mathematical Society on 20 February 1947". In: ().