Using Small-Step Refinement for Algorithm Verification in Computer Science Education*

Danijela Petrović
Faculty of Mathematics
University of Belgrade
Belgrade, Serbia
danijela@matf.bg.ac.rs

Stepwise program refinement techniques can be used to simplify program verification. Programs are better understood since their main properties are clearly stated, and verification of rather complex algorithms is reduced to proving simple statements connecting successive program specifications. Additionally, it is easy to analyze similar algorithms and to compare their properties within a single formalization. Usually, formal analysis is not done in an educational setting due to complexity of verification and a lack of tools and procedures to make comparison easy. Verification of an algorithm should not only give a correctness proof, but also better understanding of an algorithm. If the verification is based on small step program refinement, it can become simple enough to be demonstrated within the university-level computer science curriculum. In this paper we demonstrate this and give a formal analysis of two well known algorithms (Selection Sort and Heap Sort) using the proof assistant Isabelle/HOL and program refinement techniques.

1 Introduction

Correctness of computer systems is critical in todays information society. Computer systems are present in many forms all around us and often we do not even realize it. In many areas, errors in computer systems both on the side of hardware and software may cause significant financial and health costs [18]. While hardware errors are rather rare, software errors are very common and their presence in large projects is generally accepted as unavoidable. Furthermore, ensuring the correctness of the software part is often a greater problem than that of the underlying hardware. In recent years many flaws in software caused increasing effort in investigating software verification.

General program verification techniques. Program testing can reveal presence of errors, but not their absence. Software has to be executed to see how it behaves. Manual inspection of complex software is error prone and costly. Formal program verification is the process of proving that a computer program meets its specification which formally describes the expected program behavior. Early results date back to 1950s and pioneers in this field were A. Turing and J. McCarthy. In the late 1960s R. Floyd introduced equational reasoning on flowcharts for proving program correctness [9] and T. Hoare introduced axiomatic semantics for programming constructs [12]. In order to prove correctness of a program, it has to be formalized in some meta-theory so its properties can be analyzed in a rigorous mathematical manner. In order to achieve the desired highest level of trust, formalization in a classical pen-and-paper fashion is not satisfactory and, instead, a mechanized and machine-checkable formalization is preferred.

*This work was partially supported by the Serbian Ministry of Science grant 174021

Submitted to:
ThEdu'14

There are many tools for automated program verification. Many of early results in mechanical program verification were carried out by Boyer and Moore using their theorem prover. Today, verification uses SMT solvers [4],[1] and systems such as ESC/Java [8], CBMC [7], Astrée [5], PEX [19], KLEE [6], Calysto [2] and there are many more that are less used. Some are specialized for verification of C-programs [17]. However these tools still require significant annotations from the programmer to construct a proof. Many of them cannot verify complex systems. Several tools attempt to uncover design flaws using test vectors to examine specific executions of a software system. Some tools, on the other hand, can check the behavior of a design for all input vectors.

Proving full functional correctness of a program has been and still is a great challenge for fully automated systems, and for this purpose, usually an interactive approach is applied. Formalized mathematics and interactive theorem provers (sometimes referred to as proof assistants) have made great progress in recent years. Many classical mathematical theorems have been formally proved and proof assistants have been intensively used in hardware and software verification. Theorem provers that are most commonly used for program verification nowadays are Isabelle [16], Isabelle/HOL [15], Coq [3], PVS [13], HOL Light [11], etc.

Using program verification within computer science education. Program verification is usually considered to be too hard and long process that acquires good mathematical background. A verification of a program is performed using mathematical logic. Having the specification of an algorithm inside the logic, its correctness can be proved again by using the standard mathematical apparatus (mainly induction and equational reasoning). These proofs are commonly complex and the reader must have some knowledge about mathematical logic. The reader must be familiar with notions such as satisfiability, validity, logical consequence, etc. Any misunderstanding leads into a loss of accuracy of the verification. These formalizations have a common disadvantage, they are too complex to be understood by students, and this discourages students most of the time. Therefore, programmers and their educators rather use traditional (usually trial-and-error) methods.

However, many authors claim that nowadays education lacks the formal approach and it is clear why many advocate in using proof assistants [14]. This is also the case with computer science education. Students are presented many algorithms, but without formal analysis, often omitting to mention when algorithm would not work properly. Frequently, the center of a study is implementation of an algorithm whereas understanding of its structure and its properties is put aside. Software verification can bring more formal approach into teaching of algorithms and can have some advantages over traditional teaching methods.

- Verification helps to point out what are the requirements and conditions that an algorithm satisfies (pre-conditions, post-conditions and invariant conditions) and then to apply this knowledge during programming. This would help both students and educators to better understand input and output specification and the relations between them.
- Though a program might work in general, it can happen that it does not work for some inputs and students must be able to detect these situations and to create software that works properly for all inputs.
- It is suitable to separate the abstract algorithm from its specific implementation. Students can compare properties of different implementations of the same algorithms, to see the benefits of one approach or another. Also, it is possible to compare different algorithms for the same purpose (for example, for searching element, sorting, etc.) and this could help in overall understanding of algorithm construction techniques.

Therefore, lessons learned from formal verification of an algorithm can improve someones style of programming.

Modularity and refinement. The most used languages today are those who can easily be compiled into efficient code. Using heuristics and different data types makes code more complex and seems like a perplex mixture to novices of many new notions, definitions, concepts. These techniques and methods in programming makes programs more efficient but are rather hard to be intuitively understood. On the other hand, modularity is a highly accepted principle in nowadays programming. Adhering to this principle enables programmer to easily maintain the code.

The best way to apply modularity on program verification and to make verification flexible enough to add new capabilities to the program keeping current verification intact is *program refinement*. Program refinement is the verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. It starts from the abstract level, describing only the requirements for input and output. Implementation is obtained at the end of the verification process (often by means of code generation [10]). Stepwise refinement allows this process to be done in stages. There are many benefits of using refinement techniques in verification.

- It gives a better understanding of programs that are verified.
- The algorithm can be analyzed and understood on different level of abstraction.
- It is possible to verify different implementations for some part of the program, discussing the benefits of one approach or another.
- It can be easily proved that these different implementation share some same properties which are proved before splitting into two directions.
- It is easy to maintain the code and the verification. Usually, whenever the implementation of the program changes, the correctness proofs must be adapted to these changes, and if refinement is used, it is not necessary to rewrite the entire verification, just add or change a small part of it.
- Using refinement approach makes an algorithm suitable for a case study in teaching. Properties and specifications of the program are clearly stated and it helps teachers and students better to teach or understand them.

We claim that the full potential of refinement comes only when it is applied stepwise, and in many small steps. If the program is refined in many steps, and data structures and algorithms are introduced one-by-one, then proving the correctness between the successive specifications becomes easy. Abstracting and separating each algorithmic idea and each data-structure that is used to give an efficient implementation of an algorithm is a very important task in programmer education.

As an example of using small step refinement, in this paper we analyze two widely known algorithms, Selection Sort and Heap Sort. There are many reasons why we decided to use them.

- They are largely studied in different contexts and they are studied in almost all computer science curricula.
- They belong to the same family of algorithms and they are good example for illustrating the refinement techniques. They are a nice example of how one can improve on a same idea by introducing more efficient underlying data-structures and more efficient algorithms.
- Their implementation uses different programming constructs: loops (or recursion), arrays (or lists), trees, etc. We show how to analyze all these constructs in a formal setting.

There are many formalizations of sorting algorithms that are done both automatically or interactively and they undoubtedly proved that these algorithms are correct. In this paper we are giving a new approach in their verification, that insists on formally analyzing connections between them, instead of only proving their correctness (which has been well established many times). Our central motivation is that these connections contribute to deeper algorithm understanding much more than separate verification of each algorithm.

The paper gives a clear picture of central ideas relevant for verification by means of small step refinement. We give all definitions, but some proofs and implementations are omitted, and can be found in Isabelle/HOL formalization[1].

2 Bibliography

References

[1] Alessandro Armando, Jacopo Mantovani & Lorenzo Platania (2009): *Bounded model checking of software using SMT solvers instead of SAT solvers*. International Journal on Software Tools for Technology Transfer 11(1), pp. 69–83, doi:10.1007/11691617_9.

[2] Domagoj Babic & Alan J Hu (2008): *Calysto*. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, IEEE, pp. 211–220, doi:10.1145/1368088.1368118.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, doi:10.1007/978-3-662-07964-5.

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman & Yunshan Zhu (2003): *Bounded model checking*. Advances in computers 58, pp. 117–148, doi:10.1016/S0065-2458(03)58003-2.

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival (2002): *Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software*. In: The Essence of Computation, Springer, pp. 85–108, doi:10.1007/3-540-36377-7_5.

[6] Cristian Cadar, Daniel Dunbar & Dawson R Engler (2008): *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.* In: OSDI, 8, pp. 209–224. Available at `https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar_html/`.

[7] Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A tool for checking ANSI-C programs*. In: Tools and Algorithms for the Construction and Analysis of Systems, Springer, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[8] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe & Raymie Stata (2002): *Extended static checking for Java*. In: ACM Sigplan Notices, 37, ACM, pp. 234–245, doi:10.1145/543552.512558.

[9] Robert W Floyd (1967): *Assigning meanings to programs*. Mathematical aspects of computer science 19(19-32), p. 1, doi:10.1007/978-94-011-1793-7_4.

[10] Florian Haftmann (2008): *Code generation from Isabelle/HOL theories*. doi:10.1.1.278.5329. Available at `http://www.cl.cam.ac.uk/research/hvg/Isabelle`.

[11] John Harrison (2009): *HOL light: An overview*. In: Theorem Proving in Higher Order Logics, Springer, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[12] Charles Antony Richard Hoare (1969): *An axiomatic basis for computer programming*. Communications of the ACM 12(10), pp. 576–580, doi:10.1145/357980.358001.

[1] Available http://www.matf.bg.ac.rs/ danijela/ssort.zip

[13] César A Munoz & Ramiro A Demasi (2012): *Advanced Theorem Proving Techniques in PVS and Applications*. In: *Tools for Practical Software Verification*, Springer, pp. 96–132, doi:10.1007/978-3-642-35746-6_4.

[14] Tobias Nipkow (2012): *Teaching semantics with a proof assistant: No more LSD trip proofs*. In: *Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 24–38, doi:10.1007/978-3-642-27940-9_3.

[15] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. 2283, Springer, doi:10.1007/3-540-45949-9.

[16] Lawrence C Paulson (1994): *Isabelle: A generic theorem prover*. 828, Springer, doi:10.1007/BFb0030541.

[17] Junyan Qian & Baowen Xu (2007): *Formal verification for C program*. *Informatica* 18(2), pp. 289–304. Available at `http://www.mii.lt/informatica/pdf/INFO672.pdf`.

[18] Gregory Tassey (2002): *The economic impacts of inadequate infrastructure for software testing*. *National Institute of Standards and Technology, RTI Project* 7007(011). Available at `http://www.nist.gov/director/planning/upload/report02-3.pdf`.

[19] Nikolai Tillmann & Jonathan De Halleux (2008): *Pex–white box test generation for. net*. In: *Tests and Proofs*, Springer, pp. 134–153, doi:10.1007/978-3-540-79124-9_10.