Specification and Verification in Introductory Computer Science

Frank Pfenning
Research Proposal
MSR-CMU Center for Computational Thinking
October 2010

Project Description

One of the fundamental concepts in computational thinking is the separation of *what* from *how* and the relationship between them. Yet today's introductory education in computer science concentrates almost exclusively on *how*. Even when specifications are mentioned, they remain informal and are rarely actively practiced by the students.

> We contend that introductory computer science education can be dramatically improved through the introduction of explicit specifications into the programming process. This includes pre- and post-conditions for functions, loop invariants, and data structure invariants.

In an on-going freshmen-level pilot course on *Principles of Imperative Computation* at Carnegie Mellon University, we have successfully introduced a number of computational thinking concepts, including the use of specifications. At present, these specifications are used only for dynamic checking, and only when the code is compiled with a particular flag. While this has already helped the students significantly in designing and debugging their code, we believe it falls far short of what could be accomplished with tools for static checking, giving the students immediate feedback during the programming process.

We propose to investigate a number of questions surrounding the use of specifications in introductory computer science education. While we can rely on and exploit recent progress in the field of formal verification, its use in introductory computer science courses creates a number of new research questions and opportunities.

- Can we use specifications to ensure correctness for portions of the program, thereby guiding the development and debugging process at the introductory level?
- Can we provide students with meaningful counterexamples to specifications that are not satisfied?
- Can we devise a practical system of blame assignment when a combination of static and dynamic checking indicates errors in the program?
- Can we exploit specifications to guide generation of test cases?

We envision that this research will be carried out with the C0 subset of the C language that we have specifically designed for the purpose of teaching introductory imperative programming and computational thinking[1] and which is currently in use in the pilot course mentioned above. In order to reduce the number of concepts that students need to learn, specifications do not use general logical constructs such as quantifiers, but are expressed as ordinary, boolean functions without side effects. This creates some additional research questions.

[1]Thanks to a 2009–10 grant from the CMU-MSR Center for Computational Thinking

- Can we accomplish our educational goals without introducing any explicit logical language or formal rules for reasoning?
- Can we design and build effective tools as sketched above using only functions definable within a simple imperative language?

Finally, there are some considerations particularly pertinent to the educational context in which this research takes place. On the positive side, a large number of programs from novice programmers with varying backgrounds and skill levels will be available for calibration and experimental evaluat. On the negative side, students have limited experience which puts strict usability requirements on any tools we develop. We also propose to consider the following educational research questions:

- Can we statistically demonstrate the effectiveness of our course, language, and tool design?
- Do the lessons and techniques learned transfer to later courses within an undergraduate computer science curriculum? What specific measures can we take to ensure students' preparation for follow-on courses, specifically those traditionally using imperative or object-oriented programming languages?

Project Context

How do we educate the next generation of scientists and engineers in computer science? We believe that computational thinking must be a central part of such an education, going beyond "programming" in itself. Only the combination and relationships between computational thinking, programming, and algorithms gives students the breadth of techniques and depth of understanding to solve the computational problems their disciplines will face in this century. Traditional introductory computer science education falls far short. We believe that deeper reasoning will be crucial, as well as languages and tools that support the teaching of such reasoning. This is precisely what the proposed project is aiming at. Besides its intrinsic value, we also believe that the increasing use of specifications in the computer industry itself has created the need for our graduates to have a strong working knowledge in this area.

Collaboration with MSR

I have been in touch with researchers in the MSR RiSE group, specifically Rustan Leino and Nikolaj Bjørner, who both are quite interested in a proposed collaboration. Some of the original inspiration for the course and approach came from Manuvir Das, now at MSR India. Other researchers at MSR India in the Rigorous Software Engineering group like Sriram Rajamani and Aditya Nori may also be interested, as might Byron Cook from MSR Cambridge.

Budget Information

I am requesting $100K per year for two years, starting January 1, 2011. Under the assumption of no overhead, this will cover one month of summer or academic year salary for me, one graduate student (yet to be recruited), part-time programming support from an undergraduate, plus appropriate computing and travel support, including visits to Microsoft and appropriate conferences. If I am unable to recruit a graduate student for the spring semester, I propose to partially support William Lovas, a teaching postdoc, so he can devote part of his time to tool design, implementation, and evaluation.