Safety as a Metric

Matthias Felleisen and Robert "Corky" Cartwright
Department of Computer Science
Rice University
Houston, Texas

1: From Syntax to Semantics

Most software metrics measure *syntactic* qualities of a program: number of lines, number of procedure calls, *etc*. While measuring such properties may reveal problems in programs, these metrics fail to measure the essence of programs: (partial) correctness and robustness. We therefore propose to base metrics[1] on *semantic*, instead of syntactic, criteria.

To illustrate the idea of semantics-based metrics, we have built static debuggers, which are tools that detect potential run-time failures. More specifically, a static debugger analyzes programs written in safe programming languages and pinpoints those program operations that might trigger a run-time error. The following section briefly recalls what safety means for a programming language. The third section sketches how a static debugger works and the role it plays in measuring the robustness of a program. The last section discusses the use of static debuggers in the classroom, an NSF Educational Innovation Project.

2: Safe Programming Languages

Mechanical engineers design systems so that they stabilize in the least-damaging position when [a part of] the device malfunctions. For example, the bars at a railroad crossing fall to the blocking position if both the primary power-supply and the backup battery fail to provide electricity. In contrast, many existing software systems typically continue to compute after a computation step misinterprets data.

The problem is due to the wide-spread use of unsafe programming languages like C++ or Objective C. An unsafe language does not ensure that primitive program operations are applied to arguments of the proper form. For example, a program may misinterpret an array of 10 elements as a vector of size 11 and assign some value to the 11th position, corrupting whatever data is stored in the corresponding location and continuing execution using the corrupt data. Since unsafe languages do not detect the misapplication of program operations, they cannot detect malfunctions, much less respond by taking remedial action. As a result, a malfunctioning program may print outputs or control external devices based on corrupted and meaningless data values.

A safe language prevents the misinterpretation of data by language primitives. In contrast to an unsafe language, a safe language has a semantics that completely specifies the behavior of each primitive program operation. In particular, it specifies the set of data for which the operation is defined and, implicitly, those data for which it is undefined. The

[1] The word "metric" is used here in its loose sense, not with its mathematical meaning.

language implementation then ensures with a combination of compile-time and run-time checks that program data is never corrupted. If the required check is deferred until run-time, the language generates an exception if the check fails, giving the program the opportunity to take remedial action. Examples of safe languages are Java, ML, and Scheme.

Unfortunately, a program written in a safe language can only recover from potential malfunctions anticipated by the programmer. All other attempts to misinterpret data abort program execution and print a diagnostic error message. While such error detection may be sufficient in applications with a human operator (such as patient monitoring) who can manually recover from the fault, it is not helpful in applications where no human intervention is possible (such as rocket guidance).

3: A Semantics-Based Metric

We can make programs written in safe languages even more robust with static debuggers. A static debugger consists of two components: (1) a static analyzer and (2) an interaction tool. The static analyzer performs a static analysis that pinpoints all potential fault-sites for a software system. The interaction tool presents the results of the static analysis, especially the potential fault-sites. Also, upon demand, the interaction tool explains the reasoning of the analyzer.

In practice, a static debugger analyzes a program using a logic that conservatively approximates the language's semantics. At a minimum, the logic should be as strong as the static type checker (if any); in general, it will be stronger. Using the results of the analysis, the static debugger isolates those program points for which it cannot prove that the language primitives are applied correctly. The interaction tool then leads the programmer through an examination of the isolated operations. Using the explanatory capabilities of the tool, the programmer must determine whether the static debugger discovered a bug or whether proving the soundness of the primitive operation was impossible due to a weakness in the program analysis. In the former case, the programmer should correct the program; in the latter case, the programmer may wish to rewrite the program, use a stronger proof system, or document why the fault cannot occur.

Roughly speaking, static debugging measures the robustness of a program. The more fault sites a programmer explains or eliminates from a program, the more likely it is that the program won't produce a run-time error. The measure is not a metric in the mathematical sense because it is impossible to assign a numeric value to the quality of a programmer's explanations. Still, by forcing the programmer to think about potential run-time problems in a program, static debugging improves the quality of software in a measurable manner.
Note: Static analysis is *useful* but not *meaningful* for unsafe languages. While an analysis of a program in an unsafe language may reveal potential bugs, it cannot guarantee that the data in memory is coherent according to any high-level criteria. ∎

Over the past ten years, the Rice Programming Languages Team has built three static debuggers for various dialects of Scheme. The first two, which were constructed by Cartwright, Fagan and Wright [Cartwright & Fagan **ACM PLDI** 90, Wright & Cartwright **ACM POPL** 97], exploit Milner-style type unification but use an algebra of record types. The third one, by Flanagan et al. [**ACM PLDI** 96], employs a form of set-based analysis. The analysis infers the flow of data by interpreting program operations as naive set-operations. The analysis tool, dubbed MrSpidey, is an integral part of Rice's Scheme programming

environment. Our team has used MrSpidey to analyze MrSpidey (20Kloc) and large parts of DrScheme (80Kloc).
Note: Detlefs [ACM Workshop on Formal Methods in Software Practice 96] and Bourdoncle [**ACM PLDI** 93] have developed competing static debuggers for Modula-3 and Pascal, respectively. Lack of space prevents a comparison. ∎

4: Static Debugging in Courses

The authors have obtained an NSF grant to develop a sequence of courses that introduce students to the foundations of safety and the use of static debuggers. The first version of the course was taught during the 1998 fall semester. It started with an introduction to program invariants and safety, followed by a detailed discussion of automated proving such statements about programs. The static debugger was introduced through demonstrations and theoretical discussions. The lectures explained both the strengths and the weaknesses of the underlying theorem proving techniques. The discussions also covered program rewriting techniques that would overcome some of the theorem prover's weaknesses.

To reinforce the topics covered in class, students worked on a series of small homeworks. The homeworks required students (1) to analyze small programs with the static debugger; (2) to distinguish between proper bugs and false warnings in the static debugger's reports; (3) to rewrite given programs in an intelligent manner so that the static debugger could eliminate certain problem reports; and (4) to develop "perfectly robust" programs (i.e., programs that provably do not contain any potential safety violations) using the static debugger.

To strengthen students' intuition about program invariants, one homework set required students to use ACL2 [Kaufmann & Moore, **IEEE ToSE** 97], a full-fledged theorem prover for Common Lisp. The problems on the homework called for the formulation of weak and strong program invariants and for an ACL2-based proof. Finally, to teach the importance of stating safety invariants at module boundaries, we also assigned one non-trivial project: a type-checker and an interpreter for a sequential subset of Java. The solutions were between 3,000 and 6,000 lines of code.

Based on the students' homeworks and their written reponses to a dozen interview questions, the lectures and the homeworks were a full success. Students learned to appreciate safe languages, safety invariants, and the advantages of static debugging over dynamic debugging. The ACL2 assignment proved to be invaluable in strengthening students' intuitions about the weaknesses and costs of automatic theorem proving. The project, however, was only a partial success. While most of the teams were able to produce useful safety specifications for their modules, they did not succeed in debugging them with MrSpidey. The size of the project was overwhelming given the available amount of time, and the homeworks had not sufficiently prepared students for inspecting programs with more than 300 lines. For future editions of the course, we will develop a revised, smaller project with more specific goals.

Acknowledgement: The work was partially supported by NSF and Texas ATP.

References: Papers authored or co-authored by the authors are available at
`www.cs.rice.edu/CS/PLT/Publications`.