Heuristic Evaluation of Programming Language Features

Technical Report UCSC-SOE-11-06

February 11, 2011

Caitlin Sadowski
Computer Science Department
University of California at Santa Cruz
1156 High Street
supertri@cs.ucsc.edu

Sri Kurniawan
Computer Engineering Department
University of California at Santa Cruz
1156 High Street
srikur@soe.ucsc.edu

ABSTRACT
Usability is an important feature for programming languages. However, user studies which compare programming languages or systems are both very expensive and typically inconclusive. In this paper, we posit that discount usability methods can be successfully applied to programming languages concepts such as language features. We give examples of useful feedback received from applying heuristic evaluation to a selection of language features targeted at parallel programming.

Author Keywords
Usability, Programming Languages, Parallel Programming

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: Miscellaneous

INTRODUCTION

Parallel and concurrent programming is extremely difficult [1, 11]. Parallel and concurrent programming is also increasingly pervasive; concurrency is a key component to all reactive applications, and the recent prevalence of multicore hardware has made exploiting parallelism a key aspect of performance optimizations [1]. A plethora of language features which aid in parallel and concurrent programming exist. Because parallel and concurrent programming is so hard, evaluating and improving the usefulness of these language features could make a big difference.

As an exercise, one of the authors of this paper modified Nielsen's 10 heuristics [13], plus the 13 tradeoffs which make up the cognitive dimensions framework [7] to create a selection of heuristics for evaluating language features, and then used those modified heuristics to evaluate the language feature of machine-checkable `yield` annotations [18], described below. The issues uncovered by performing this simple usability engineering method were striking; thinking about `yield` annotations in terms of heuristics uncovered several research questions in making `yield` annotations a usable feature. We recruited four additional participants and one additional language feature to test the hypothesis that heuristic evaluation could be a useful tool for programming languages researchers.

Language Features

In this paper, we investigate using discount usability methods, specifically heuristic evaluation, to identify potential problems in language features which have been proposed to help programmers reason about multithreaded code. We focus on two proposed language features: `atomic` annotations and `yield` annotations [5, 18].

The `yield` annotations represent the points at which context switching can affect the results of executing code. For example, take a look at the following code snippet:

```
public class Example {
  int x;
  public void foo() {
    float tmp1 = x;
    /* yield; */
    float tmp2 = x;
  }
  public void bar() {
    x = 3;
  }
}
```

The "/* yield; */" annotation represents the fact that another thread could change the value of `x` at this program point, for example, by executing the `bar()` method. Note that these annotations do not change the code behaviour, they just document the points of potential thread interference.

In contrast, `atomic` annotations represent blocks of code inside which do not contain thread interference. Here is the example from above rewritten with `atomic` annotations:

```
public class Example {
  int x;
  public void foo() {
    atomic{ float tmp1 = x; }
    atomic{ float tmp2 = x; }
  }
  atomic public void bar() {
    x = 3;
  }
}
```

Roughly speaking, `yield` annotations would correspond to the points where atomic blocks meet, in code that is entirely covered by atomic blocks. As before, `atomic` annotations do not change the code behaviour.

For both types of annotations, participants were informed that programmers could use these annotations in two ways. First of all, programmers could annotate methods with `yield` or `atomic` and then analyze their annotated programs to check if the annotations were correct. Several such checkers exist in the research literature for `atomic` annotations (e.g. [5]); one `yield` annotation checkers has also been published [19]. As an alternative use, a program (or another programmer) could annotate a program with correct annotations, perhaps via a `yield` annotation inference tool [19]. Future programmers could use these annotations to reason about the code.

RELATED WORK

Usability testing has sometimes been used to inform the design of programming languages. The most notable example of this is the HANDS language for children [16]. This language involved HCI principles as an integral part of the language development process; after the language was developed, the authors performed a user study to evaluate the addition of some key language features by comparing language versions with and without the features.

Discount usability [12, 13] is a set of fast techniques for evaluating the usability of a system. These techniques include scenarios, card sorting, and heuristic evaluation. Discount usability engineering has been used in a variety of contexts, including agile development [10]. However, discount usability methods have also been criticized, particularly for not having good coverage metrics or a clear way to analyze the results for false positives or false negatives [6].

In this paper, we are running a variant of heuristic evaluation [14]. Heuristic evaluation has been successfully used in a variety of contexts, ranging from evaluating mission-critical software with a large team [3] to evaluating games [17] and has been shown to be effective at finding severe usability problems [8]. Nielsen's heuristics have also been used to categorize prior research focused on usability and novice programming systems [15].

We base some of our heuristics on the cognitive dimensions framework. This framework was originally presented as a set of conceptual categories which highlight design tradeoffs in visual programming languages [7]. These dimensions were later formulated as a questionnaire for users [2]. Other researchers have used cognitive dimensions to evaluate programming languages [4], or formatively in developing a language feature [9].

METHODOLOGY

We took Nielsen's 10 heuristics [13], plus the 13 tradeoffs which make up the cognitive dimensions framework [7], rewritten as heuristics. We updated, merged, or deleted heuristics which did not make sense in the context of a language feature. The final list of heuristics is displayed in Table 1.

Using these 11 heuristics we ran a heuristic evaluation with five participants, including one of the authors of this paper. The participants were computer science graduate students and were all experts in parallel and concurrent programming. We first had participants evaluate `yield` annotations with the heuristics, and then evaluate `atomic` annotations. All participants were familiar with research on `yield` and `atomic` annotations before the study.

Abstraction Gradient	Does this feature adequately cover the gradient between minimum and maximum levels of abstraction in a program? (e.g. the gradient from low-level code details to interface-level details)
Consistency	Does this feature have consistent meaning?
Error-Proneness	Does the notation for the language feature induce "care-less mistakes"? Is there a match between notation in the system and how similar language is used in the real world?
Hidden Dependencies	Are all dependencies clear with this feature? Could local changes have confusing global effects?
Premature Commitment	By using this language feature, do programmers have to make decisions before they have the information they need?
Progressive Evaluation	Can a partially-complete application of this feature provide feedback on "How am I doing"?
Viscosity	How much effort is required to perform a single change involving this feature?
Flexibility & Efficiency	Is this feature effective across the gradient between novice and expert programmers?
Aesthetic Design	Feature notation should be simple and concise.
Error Recovery	If there is an error in language feature usage, are precise, constructive error messages in plain language presented?
Documentation	Even though it is better if the language feature can be used without documentation, it may be necessary to provide help and documentation. Documentation should be concise, concrete, and relevant.

Table 1. Eleven Language Feature Heursitics

Taking inspiration from a study about developing a heuristic evaluation for video games [17], we asked the four non-author participants to describe strengths and limitations of using the supplied heuristics to evaluate language features. We also asked participants to identify any heuristics they found particularly useful or particularly confusing. Lastly, we asked participants whether they found problems using the heuristics they would otherwise have missed, and whether the heuristics gave them new perspectives on the language feature.

RESULTS

Participants identified 5 problems shared between `atomic` annotations and `yield` annotations; of these problems, we are not aware of prior discussion of 2 of them in the research literature. Participants identified 7 problems unique to `yield`

annotations, 5 of which are (to our knowledge) new problems. Participants also identified 5 problems unique to `atomic` annotations, 2 of which are (to our knowledge) new problems.

Participants rated the same problems at various places on the scale between cosmetic and catastrophic; we have omitted the severity rating assigned from the following discussion since ratings were very mixed and we feel that the identification of all the new problems is the major research contribution of this work. Every heuristic resulted in at least one participant identifying a problem. Because of this, we believe that the list of heuristics could be expanded, but should not be compressed.

Problems shared between atomic and yield annotations

1. Participants were concerned about other definitions of the words "atomic" and "yield" leading to confusion.

   > *"Yield is an overloaded word; not self-documenting."*

   We feel that the choice of terminology can have a big impact in understanding concepts; we recommend that possible sources of notational confusion should be discussed more in the literature.

2. Participants were concerned about the usefulness of tool feedback for missing `yield` or `atomic` annotations. We feel that this could be discussed more in the literature.

3. Participants realized that missing `yield` annotations or inaccurate `atomic` sections may lead to strange results and may not represent real bugs.

   > *"It is unclear how bugs related to inadequate yield annotations will actually manifest."*

   > *"Knowing something is non-atomic may result in making changes to an entirely separate part of the program."*

   This problem represents an area of active research.

The remaining two problems are not, to our knowledge, discussed in the research literature.

1. Participants were concerned about how `atomic` or `yield` annotations relate to evolving code. Although `yield` annotations may be useful in program evolution [18], this aspect needs to be explored further. We are not aware of research which looks at how atomic blocks of code evolve over time.

2. All participants were concerned about the lack of documentation for these features, beyond research papers.

Problems with yield annotations

Two problems identified with `yield` annotations represent general issues in annotation checking tools and so are not unique to this particular language feature.

1. Participants were concerned about the impact of false positives or negatives in the checker on the usefulness of `yield` annotations.

2. Participants were worried about users ignoring `yield` annotations.

The remaining five problems are not, to our knowledge, discussed in the research literature.

1. Participants worried about a false sense of security caused by correct yield annotations.

2. Participants identified a problem in the difference between local and global reasoning with yields.

   > *"Yields are specific to a line number; but how do you reason about methods that may contain yields inside without looking at the code?"*

3. Participants were worried that too many yields could clutter code, be confusing, or be annoying to write.

4. One participant pointed out that:

   > *"Partially correct set of yield annotations is not worth very much."*

5. One participant thought the word yield might be too minimal, and that perhaps additional information could be included in the notation.

Problems with atomic annotations

Three identified problems with atomic annotations are also discussed in the literature on yield annotations [18].

1. Atomic sections create bimodal reasoning.

   > *"Atomic forces programmers to think about code as if [it is either] in atomic or out of atomic."*

2. Participants pointed out that `atomic` annotations based on syntactic blocks are limited.

3. Participants were concerned about the global impact of atomic or non-atomic sections.

   > *"How does making one method atomic or non-atomic change the entire program behaviour?"*

The remaining two problems are not, to our knowledge, discussed in the research literature.

1. Participants wondered how `atomic` annotations could be applied at the class level.

   > *"Does an atomic interface mean a safe interface?"*

2. Participants noted that `atomic` annotations were only useful if correctly applied.

   > *"[Atomic annotations are] easy to add, hard to add well."*

Feedback

Multiple participants found that the heuristics which focused on error-proneness or progressive evaluation were particularly useful; we think these are important heuristics for programming language researchers to consider when developing language features or tools. Two participants found thinking about the abstraction gradient to be particularly confusing, but a third participant found this to be particularly useful; we believe that this heuristic was stated confusingly, and will be more useful once it is re-worded.

There was only one participant who did not strongly agree that using heuristics both helped them find new problems and gave them a new perspective. This remaining participant strongly agreed that the heuristic evaluation was a helpful exercise, but thought the process clarified his thinking rather than providing an entirely new perspective. All participants thought this would be a useful methodology for their own research.

FUTURE WORK

This initial evaluation, though small, uncovered several research issues in a short amount of time. However, the small number of participants and the ambiguity of interpretation of some of the heuristics represent threats to validity of this study. We would like to compare a larger set of language features using heuristic evaluation. We would like to investigate other possible heuristics, and develop a theory supporting the use of these future heuristics in evaluating language features. We also feel that the 0-4 rating scheme for heuristics does not quite fit the language-feature context; one participant also commented on this in their feedback survey.

We would like to further explore whether other discount usability methods can be used for programming languages features or concepts. Lastly, we would like to better understand adoption barriers within the programming languages research community which prevent similar methods from being used.

REFERENCES

1. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM (CACM)*, 52(10):56–67, 2009.

2. A. Blackwell and T. Green. A Cognitive Dimensions questionnaire optimised for users. In *Annual Meeting of the Psychology of Programming Interest Group (PPIG)*, volume 12, pages 137–152. Citeseer, 2000.

3. T. Buxton, A. Tarrell, and A. Fruhling. Heuristic Evaluation of Mission-Critical Software Using a Large Team. *International Conference on Human-Computer Interaction*, pages 673–682, 2009.

4. S. Clarke. Evaluating a new programming language. In *Workshop of the Psychology of Programming Interest Group (PPIG)*, pages 275–289, 2001.

5. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003.

6. W. Gray. Who Ya Gonna Call? You're on Your Own. *IEEE Software*, 14(4):26, 1997.

7. T. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

8. R. Jeffries, J. Miller, C. Wharton, and K. Uyeda. User interface evaluation in the real world: a comparison of four techniques. In *Conference on Human factors In computing systems (CHI)*, pages 119–124. ACM, 1991.

9. S. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. In *International Conference on Functional Programming (ICFP)*, pages 176–186. ACM, 2003.

10. D. Kane. Finding a place for discount usability engineering in agile development: throwing down the gauntlet. In *Agile Development Conference (ADC)*. IEEE, 2003.

11. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 43(3):329–339, 2008.

12. J. Nielsen. Applying discount usability engineering. *IEEE Software*, 12(1):98–100, 1995.

13. J. Nielsen and R. L. Mack, editors. *Usability inspection methods*. John Wiley & Sons, Inc., 1994.

14. J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *Conference on Human factors In computing systems (CHI)*, pages 249–256. ACM, 1990.

15. J. Pane and B. Myers. Usability issues in the design of novice programming systems. *CMU Human-Computer Interaction Institute Technical Report CMU-HCII-96-101*, 1996.

16. J. Pane, B. Myers, and L. Miller. Using HCI techniques to design a more usable programming system. In *Symposium on Human Centric Computing Languages and Environments*. IEEE, 2002.

17. D. Pinelle, N. Wong, and T. Stach. Heuristic evaluation for games: usability principles for video game design. In *Conference on Human factors In computing systems (CHI)*, pages 1453–1462. ACM, 2008.

18. J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.

19. J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.