Gradual Programming: Bridging the Semantic Gap

(Position Paper)

Bor-Yuh Evan Chang Amer Diwan Jeremy G. Siek

University of Colorado, Boulder

{evan.chang, amer.diwan, jeremy.siek}@colorado.edu

Abstract

There is no perfect programming language. Programmers must write code conforming to the idiosyncrasies of a programming language. Thus, there is often a disconnect between the intent of the developer and the meaning of the program. This semantic gap has a negative effect on programmer productivity, software reliability, and execution efficiency. In this position paper, we argue that in order to address this semantic gap, we must drastically rethink how we develop software.

1. Introduction

Language design is an exercise in trade-offs. Is the language biased towards static safety or towards expressiveness? Is the language biased towards applications that must be resilient to errors or applications that can simply terminate? This list of trade-offs goes on and on. For this reason, no language is a perfect fit for any non-trivial programming task. Consequently, there is often a discrepancy between what the programmer intends and what the code actually means, which we call a *semantic gap*.

One instance of a semantic gap is when the meaning of a program is missing behaviors that were intended by the programmer. For example, the `equals` method in Java's `Object` class [Gosling et al. 2005] takes an argument of type `Object`, and Java's type system requires that any override of the `equals` method in any subclass must also take an argument of type `Object`. Thus, the `equals` implementation for the `Matrix` class in Figure 1 takes an argument of type `Object` instead of an argument of type `Matrix`. Thus, there is less static checking than what was intended by the programmer. While there is a good reason for the restriction on argument types (allowing covariant argument types would introduce a hole in the type system), it is counter-intuitive and not what the programmer wants: the code fails to capture the programmer's intent. If a client of the `Matrix` class incidentally invokes `equals` with an object of another class, Java's type system will not catch the error at compile time.

Another example of a semantic gap is when the meaning of a program includes behaviors that were not intended by the programmer. Consider again the `equals` function in Figure 1. The programmer intends this function to return true if the corresponding elements of the two matrices are equal (as determined by `CompareFloats.same`) and false otherwise. Unfortunately, the actual code does much more: it may throw an exception if parameter `y` is a null pointer, it loops over the two matrices in a particular order (which in this case exhibits poor data locality), it may dynamically load and initialize the `CompareFloats` class, it may propagate exceptions that originate from the `get` method, the `same` method, or from initializers run during dynamic class loading.

PLDI-FIT '09 June 16, 2009, Dublin, Ireland.

```
class Matrix extends Object {
  private double data[][]; private int nrows, ncolumns;
  public double get(int i, int j) { return data[i][j]; }
  public boolean equals(Object y) {
    Matrix to = (Matrix)y;
    if (this.nrows != to.nrows
        || this.ncolumns != to.ncolumns) return false;
    for (int j = 0; j != ncols; ++j) {
      for (int i = 0; i != nrows; ++i) {
        if (!CompareFloats.same(get(i,j), to.get(i,j))
          return false;
      }
    }
    return true;
  }
}
```

Figure 1. An excerpt from a matrix class demonstrating two instances of a semantic gap.

Because of the semantic gap, the programmer's intent is hidden from both humans and tools that need to understand or transform the code. Furthermore, we hypothesize that this gap has a pervasive negative effect on programmer productivity, software reliability, and execution efficiency.

The semantic gap reduces programmer productivity by affecting the writability and readability of the code. To see this, consider again our `equals` example. The author of the `Matrix` class cannot write what she wants: she wants the `equals` method to take an argument of type `Matrix` and not of type `Object`. A reader of the `Matrix` class also suffers: she does not know what types of arguments are actually legal to the `equals` method (as opposed to what the Java type system allows).

The semantic gap degrades software reliability by crippling tools designed to check and improve the safety of programs. In particular, a tool that verifies safety properties of programs needs to consider all the possibilities allowed by the code instead of just the possibilities that are consistent with the programmer's intent. For example, the `equals` method cannot cause an `ArrayIndexOutOfBoundsException`; however, a conservative tool cannot prove this property because `get` may be overridden by a dynamically loaded subclass of `Matrix`. In general, the semantic gap causes tools to produce overly conservative and thus less useful results.

Finally, the semantic gap diminishes program efficiency by hindering program transformations and specifically compiler optimizations. Like an analyzer, an optimizer also needs to consider all the possibilities allowed by the code instead of the possibilities that are consistent with the programmer's intent. For example, in the `equals` method, an optimizer could improve performance by exchanging the order of the two loops and thus iterate over the rows of the matrix instead of the columns. However, the body

of the loop contains potential side effects (e.g., `get` can throw an `ArrayIndexOutOfBoundsException`), and reordering the loops would change the order of these side effects. Thus, an optimizing compiler will not transform these loops even though such a transformation would be consistent with the programmer's intent.

For these reasons, we argue that we must rethink how we develop software. Specifically, we should focus on programming languages, supporting IDEs, and methodologies that enable programmers to express their intent naturally and compactly. We envision a new methodology and tool chain based on *gradual programming*. With this approach, a programmer starts with a program that best captures her intent. However, such a program may not provide enough information to the development tool, such as the compiler. The development environment then guides the programmer in filling in the missing information; specifically, in situations where it matters, the development environment interacts with the user to obtain the additional information that it needs. Thus, the programmer ultimately ends up with a program that captures the exact behavior she intended.

Our vision is indeed a radical departure from common wisdom. In essence, we are advocating developing programs in a family of languages with varying semantics. Then, part of the development process involves nailing down the precise semantics of the program. Such a position is not without significant challenges and possible pitfalls, which we outline in Section 2. We then sketch some potential solutions to these issues in Section 3.

2. But language design is for experts

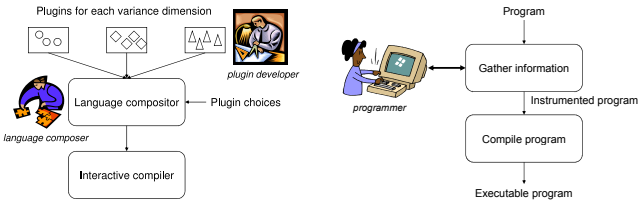
One possible counterargument to our proposal is that while program development is hard, programming language design is even harder. For example, it is widely accepted that programming language design is not compositional. In other words, many problems arise not from a single feature but from a combination of features. For instance, a famous example in the history of programming languages is the combination of type inference with parametric polymorphism and imperative references (e.g., in the dialects of ML). In particular, type safety can be subverted by allowing values of one type to be stored into reference cells and read out as another type. Modern ML dialects maintain type safety by imposing what is known as the value restriction [Wright 1995]. Thus, a key challenge is that we need to ensure that the program developer's choices work consistently together.

Semantic gaps arise because any given programming language offers only a fixed semantics for a given aspect of the language and that semantics may or may not fit the intention of the programmer. We refer to these aspects as *semantic dimensions* (or *dimensions* for short). For instance, one dimension may be "what happens when we dereference a null pointer?". This dimension has at least two reasonable variants: throw a null-pointer exception, which the program may handle, or terminate the program. Existing languages force one of these two variants on all programs in the language.

We are not in fact arguing that program developers should also be language designers. Rather a gradual programming system may involve several kinds of users, or *roles*, that require varying levels of expertise in language design and semantics. For example, on a particular software project, one language expert may be responsible for deciding which semantic dimensions are allowed to vary. With the ability to put constraints on the semantics that may vary, a development environment for gradual programming becomes more feasible.

3. Roles and tool support

Figure 2 gives an overview of how users interact with a potential gradual programming system. Users play three distinct roles: the



(a) Instantiating an interactive gradual programming compiler with language plugins.

(b) Gradual programming with an interactive compiler.

Figure 2. A potential gradual programming system.

plugin developer, the *language composer*, and the *programmer*. The *plugin developer* creates variants for semantic dimensions by implementing plugins that conform to the interfaces for the respective dimensions. Besides implementing the core functionality of a variant (e.g., generating code to throw exceptions on a null pointer dereference), plugins provide information that is useful for determining its interactions with plugins for other dimensions. The plugin developer is expected to be an expert in compiler construction. The *language composer* defines a language by selecting plugins. For example, before starting a new project, a company may employ a language composer who picks plugins for each dimension. If the language composer picks more than one plugin for a dimension, then the language will contain a choice that the programmer must make (e.g., parameter passing mode). The language composer must pick plugins that work well with each other. Though potentially assisted by program analysis tools, the language composer must be an expert in programming languages. The *programmer* uses the language selected by the language composer. If the language has more than one plugin for a given dimension, then there are two possibilities: (i) if the compiler can determine that the different possibilities all have the same observable behavior, then the compiler can pick one automatically (e.g., it may pick one that enables the most optimizations); (ii) if the compiler finds that different plugins behave differently for the program, then the compiler interacts with the programmer to determine which of the possible plugins to use. Such choices should be reflected in the program so that they are maintained as the program evolves.

We argue that this stratification of users provides a path to addressing the challenges while affording the flexibility we want with gradual programming. In particular, programmers have the ability to better specify their intent by deciding what "language semantic" plugins apply at which points in their program. At the same time, we leave the definition of dimensions and variants, the implementation of plugins, and the checking of problematic interactions to language or compiler design experts (i.e., the plugin developer and the language composer).

4. Conclusion

The semantic gap between what the programmer intends and what the code actually means significantly impedes our efforts to improve programmer productivity, software reliability, and execution efficiency. To address the semantic gap, we argue that we must radically rethink how we develop software towards a model, methodology, and system for gradual programming.

References

James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.