Ph.D. Thesis Proposal: A More General Programming Language Supporting Separation Of Concerns

Doug Orleans

March 13, 2002

Abstract

Separation of concerns is a problem solving technique that should be well supported by programming languages. Aspect-oriented programming languages purport to allow the modular expression of concerns that can not be separated in other currently popular programming langugaes. In particular, AspectJ supports the modularization of concerns that would otherwise cut across the modularization supported by Java. I conjecture that it can be formally shown that AspectJ and other AOP languages that extend Java are more expressive than Java (given some calculi that approximate the languages). I further conjecture that AspectJ is more general than Java—that is, some constructs in Java can be expressed as syntactic sugar in AspectJ—and I propose a new language called Fred that is more general than AspectJ, therefore providing better support for separation of concerns.

1 Introduction

In 1976, Edsger W. Dijkstra described a technique for problem solving [1, page 211]: "study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects"; he dubbed this technique **separation of concerns**. It is not easy to decide how a problem should be decomposed into separate concerns, but realizing that a decomposition is needed is an important first step in solving the problem.

Ideally, the decomposition of a problem into separate concerns should lead directly to the modularization of a program that solves the problem—that is to say, each program module should embody one (and only one) concern of the problem. The more this is true, the easier the program is to be understood, modified, or reused; the less this is true, the more the program tends to be tangled, brittle, and overspecialized.

In practice, this correspondence is limited by the constructs available in the programming language. In recent years, it has become clear that many concerns that one would wish to separate out from a problem cannot be properly modularized using the constructs of any currently popular programming language, because the concerns cut across the modularization that is supported by the existing constructs. This realization has led to the design of **aspect-oriented programming (AOP)** [9] languages that support the modularization of crosscutting concerns.

AOP language design is an active area of research; some of the more prominent languages under development are AspectJ [8], HyperJ [12], ComposeJ [13], DemeterJ [6], and the language of aspectual components [10]. They all purport to address the problem of crosscutting concerns, by extending Java [5] with new constructs for expressing concerns that could not be expressed modularly in plain Java.

The claim that these languages are more expressive than Java is an informal one, usually based on a set of illustrating examples. It would be valuable to prove this claim formally, based on the syntax and semantics of the languages. Even more valuable would be to prove expressiveness relations between different AOP languages. This could lead towards the definition of new languages which are even more expressive, which would allow a better correspondence between program modules and problem concerns.

While the expressive power of programming languages is usually thought of only in terms of computability—a language can either be universal (Turing-complete) or not—Felleisen [3] developed a formal notion of comparative expressiveness that can distinguish the expressive power of two universal languages. His system is based on the intuition that a language $\mathcal{L}$ that contains a superset of the constructs of another language $\mathcal{L}'$ is more expressive than $\mathcal{L}'$ if a behavior-preserving translation from $\mathcal{L}$ to $\mathcal{L}'$ requires a global reorganization of the entire program.

AOP languages that extend another language are often thought of in terms of "weaving" pieces of code together into the base language, which matches this notion of global reorganization. I conjecture, then, that AspectJ and the other AOP languages listed above are all more expressive than Java, according to Felleisen's definition of expressiveness. In order to show this formally, I intend to start with minimal core calculi representing the basic features of Java, AspectJ, etc., in the manner of Featherweight Java (FJ) [7], and continue with successively larger approximations of the full languages. Section 2 outlines this approach in further detail.

An interesting feature of AspectJ is that not only does it add a new construct for defining crosscutting behavior, namely **advice**, but this construct can also be used to express ordinary Java methods. The effect is that AspectJ is no more expressive than the subset of AspectJ that does not include methods. While programmers will probably not want to program without methods, the fact that methods are inessential to AspectJ makes it easier to model the language: one fewer construct is needed in its formal definition, because methods can be replaced with syntactic sugar on top of advice. Section 3 formalizes this concept into a notion of relative generality of languages, and outlines an approach for showing that AspectJ is more general than Java.

A natural next question to ask is whether there are languages that support separation of concerns that are even more general than AspectJ. I have developed a small language called Fred [11], based on predicate dispatching [2], that can express many of the constructs in AspectJ as macros. Since predicate dispatching is a generalization of multiple dispatch, which itself is a generalization of the single dispatch object-oriented model of AspectJ and Java, it stands to reason that Fred is more general than AspectJ. Section 4 describes Fred and outlines an approach for showing that it is more general than AspectJ.

In summary, my proposed thesis is that aspect-oriented programming languages, while more general than typical object-oriented programming languages, can be made even more general, providing better support for separation of concerns. Section 5 outlines the research required to complete the dissertation.

2 Relative Expressiveness of Languages

Felleisen provides the following definitions [3]:

Definition 2.1 A **programming language** $\mathcal{L}$ consists of

- a set of $\mathcal{L}$-phrases, which is a set of freely generated abstract syntax trees (or **terms**), based on a possibly infinite number of function symbols $F, F_1, \ldots$ with arities $a, a_1, \ldots$;
- a set of $\mathcal{L}$-programs, which is a non-empty, recursive subset of the set of phrases; and
- a semantics, $eval_{\mathcal{L}}$, which is a recursively enumerable predicate on the set of $\mathcal{L}$-programs. If $eval_{\mathcal{L}}$ holds for a program P, the program **terminates**.

The function symbols are referred to as **programming constructs** or **programming facilities**.

Definition 2.2 A programming language $\mathcal{L}$ is a **conservative extension** of a language $\mathcal{L}'$ if

- the constructors of $\mathcal{L}'$ are a subset of the constructors of $\mathcal{L}$ with the difference being $\{F_1, \ldots, F_n, \ldots\}$, which are not constructors of $\mathcal{L}'$;
- the set of $\mathcal{L}'$-phrases is the full subset of $\mathcal{L}$-phrases that do not contain any constructs in $\{F_1, \ldots, F_n, \ldots\}$;
- the set of $\mathcal{L}'$-programs is the full subset of $\mathcal{L}$-programs that do not contain any constructs in $\{F_1, \ldots, F_n, \ldots\}$; and
- the semantics of $\mathcal{L}'$, $eval'_{\mathcal{L}}$, is a restriction of $\mathcal{L}$'s semantics, i.e., for all $\mathcal{L}'$-programs P, $eval'_{\mathcal{L}}(P)$ holds if and only if $eval_{\mathcal{L}}(P)$ holds.

Conservatively, $\mathcal{L}'$is a **conservative restriction** of $\mathcal{L}$. To emphasize the constructors on which the restriction and extension differ, we write $\mathcal{L}' = \mathcal{L}\backslash\{F_1, \ldots, F_n, \ldots\}$ and $\mathcal{L} = \mathcal{L}' + \{F_1, \ldots, F_n, \ldots\}$.

Definition 2.3 Let $\mathcal{L}$ be a programming language and let $\{F_1, \ldots, F_n, \ldots\}$be a subset of its constructors such that $\mathcal{L}' = \mathcal{L}\backslash\{F_1, \ldots, F_n, \ldots\}$ is a conservative restriction. The programming facilities $F_1, \ldots, F_n, \ldots$ are **eliminable** if there is a recursive mapping φ from $\mathcal{L}$-phrases to $\mathcal{L}'$-phrases that satisfies the following conditions:

E1 $\varphi(e)$ is an $\mathcal{L}'$-program for all $\mathcal{L}$-programs e;

E2 $\varphi(F(e_1, \ldots, e_a)) = F(\varphi(e_1), \ldots, \varphi(e_a))$ for all facilities F of $\mathcal{L}'$, i.e., φ is homomorphic in all constructs of $\mathcal{L}'$; and

E3 $eval_{\mathcal{L}}(e)$ holds if and only if $eval'_{\mathcal{L}}(\varphi(e))$ holds for all $\mathcal{L}$-programs e.

We also say that $\mathcal{L}'$ **can express the facilities** $F_1, \ldots, F_n, \ldots$ **with respect to** $\mathcal{L}$.

It is clear that AspectJ is a conservative extension of Java. The goal, then, is to show that there are facilities in AspectJ that Java cannot express. One way to do this would be to find an AspectJ program that terminates and show that no structure-preserving translations to Java programs would terminate. Here is such a program (assuming it is invoked from the command line as `java Main`):

```
class Main {
  public static void main(String args[]) { while (true); }
}
aspect Terminate {
  void around(): call(* main(*)) { }
}
```

While I will not formally prove that this program meets our criteria, here is an informal argument: since any translation must preserve the structure of all constructs in Java, the class definition `Main` must be preserved verbatim. Since there is no way in Java to avoid invoking the `main` method of the class specified on the command line, the translated program must execute the `while` loop and never terminate.

3 Relative Generality of Languages

Definition 3.1 A language $\mathcal{L}$ is **more general** than a conservative restriction $\mathcal{L}'$ if $\mathcal{L}'$ cannot express some facilities in $\mathcal{L}$ and one or more non-eliminable facilities in $\mathcal{L}'$ are eliminable from $\mathcal{L}$.

The goal here is to show that there are facilities in Java that are not eliminable but are eliminable from AspectJ. In particular, concrete method definitions are not eliminable from Java, but can be translated in AspectJ to advice:

```
class Main {
  void foo() { ... }
}
```

can be translated to

```
class Main {
  abstract void foo();
  void around(): target(Main) && call(void foo()) { ... }
}
```

4 Fred: A More General AOP Language

- behavioral model of Fred (branches, predicates, decision points).
- syntactic sugar for OO-style methods, AspectJ-style pointcuts and around advice.
- prototype implementation in MzScheme [4].

5 Research Plan

To be added to Fred:

- structural model: separately-specified fields and inheritance (like AspectJ static crosscutting, e.g. introductions and "declare parents").
- variable binding in predicates (like AspectJ context exposure)
- parameterized encapsulation: lexical scoping, plus bundles (like units). Better than using classes for parameterized encapsulation (like AspectJ abstract pointcuts).
- syntax for classes, aspects, hyperslices, composition filters, traversals & visitors, aspectual collaborations. Comparative examples.
- customizable branch precedence
- more efficient implementation– static analysis
- formal semantics of core language
- development environment: browser that can show which branches may be applicable for a message send, and their precedence.

References

[1] E. W. Dijsktra. *A Discipline of Programming*. Prentice-Hall, 1976.

[2] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20–24 1998.

[3] M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432, pages 134–151. Springer-Verlag, New York, NY, 1990.

[4] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[6] G. Hulten, K. Lieberherr, J. Marshall, D. Orleans, and B. Samuel. *DemeterJ User Manual*. `http://www.ccs.neu.edu/research/demeter/`.

[7] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 1997.

[10] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual collaborations for collaboration-oriented concerns. Technical Report NU-CCS-01-08, College of Computer Science, Northeastern University, Boston, MA 02115, Nov. 2001.

[11] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands, April 2002.

[12] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000.

[13] J. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the Java language. MSc. thesis, Dept. of Computer Science, University of Twente, Enschede, the Netherlands, December 1999.