Research Agenda

Andrew P. Black

November 6, 2004

1 My Research Goals

My research over the last 27 years — since I started my doctoral work in 1977 — has been driven by a single idea: programming is difficult. What I mean by this, of course, is that *I* find programming difficult. My goal as a computer science researcher has been, and remains: to make programming easier.

I recognize that not everyone finds programming difficult. Indeed, many of my peers seem to find programming very easy, and positively revel in being able to master the complexity of an application in a write-only programming language that makes full use of a large set of support libraries to execute a complicated distributed algorithm. My research is not directed at these people. It is instead focussed on those who find programming difficult, who find the process of extracting the meaning from the code hard, and who find the idea of modifying a large existing code base intimidating.

In 1977 it was clear that the programmer's most important tool was the programming language, and that a major route to making programming easier was to make better programming languages. But it follows from my driving idea that building a better programming language was a design activity, not mathematics. To put it another way: the test of whether one language or language feature was better than another had less to do with mathematical elegance than with tailoring the language to the way that people think. Mathematics, like tradition, has a vote, but not a veto. While it is true that a language construct with an ugly mathematical semantics is unlikely to be easy for a programmer to understand or use, the converse is not always true: a language construct that has a simple and elegant mathematical semantics is not necessarily easy to use or understand.

A good example of this phenomenon is the popularity of objects in programming. It is clear to anyone who has compared the semantics of a declarative and an object-oriented language that the declarative language is far simpler. But this does not necessarily make it easier for the average programmer to use. The object-based language makes it possible to program by simulation of the stateful objects that exist in the real world, which is simpler for most programmers to grasp because their understanding of the real world is much deeper than their understanding of mathematics.

In a similar way, inheritance, which has a highly complex semantics, remains easier for most

programmers to think about than parameterization, in spite of the fact that parameterization is mathematically much simpler. I believe that this is because people are much better at moving from the concrete to the abstract than from the abstract to the concrete. A well-written public method in an inheritance hierarchy starts with a concrete algorithm, nicely factored into subsidiary methods (called *hook* methods), whose names convey an approximation of what they contribute to the goal of the public method. But these hook methods also have implementations, which in effect provide concrete examples of what overridings of that hook method should achieve. In subclasses, the hook methods work like parameters, allowing programmers to change the behavior of the public method, sometimes in surprising ways. The same effect could have been obtained by writing a highly parameterized function instead of the original public method, and then providing values for the parameters instead of defining the hook methods in the sub-classes. Conceptually, this alternative is simpler. But the object-based approach makes task of understanding the goal and methodology of the public method immeasurably easier, because its description incorporates *concrete examples* of the actions of the hook methods.

2 Early Work: 1977–1986

Given this view of the role of programming language research, it is clear that having programmers actually *use* my ideas in real applications is important. While a doctoral student I collaborated with a consulting engineer who had won a contract to produce a program that was both *executable* and provided a *readable* definition of a complex quantity surveying calculation. Readability was vital because this program would be incorporated into the contract and payment process for major public building construction projects. My doctoral thesis was an examination of the effect of a programming language construct (Exception Handling) on the readability of programs.

After Oxford, I took up a faculty position at the University of Washington. One of the reasons for selecting Washington was that UW had just started a large system software project (the Eden Project), and I would have the opportunity to provide programming tools for this community.

This turned out to be fascinating and productive experience. My involvement with the Eden project brought me into contact with distributed systems just at the time that distribution was emerging as a field of study. The Eden Programming Language, which I designed and implemented with the help of a number of research students, incorporated the first object-oriented RPC-like system: the language implementation took care of all of the mechanical details of sending messages to remote objects and receiving the replies. Publications 50-53 on my vita represent this work. Two of these publications (50 and 53) were presented at SOSP, the preeminent operating systems conference then as now, and much more selective than most journals. Paper 52 was presented at ICSE, also a very discriminating venue, whereas 51 was published in the leading Software Engineering journal (IEEE TSE).

Building on our experiences with Eden, I, Hank Levy, and two Ph.D. students, Norman Hutchinson and Eric Jul, undertook the design and implementation of the Emerald programming language. If one measures impact by publication in the best venues and by citation counts, this is some of my most significant work. The initial OOPSLA paper describing Emerald Objects [47] is cited 146 times; OOPSLA is the pre-eminent venue for object-oriented research in the USA. The

paper formalizing the type system [46] was published in IEEE TSE and is cited 92 times. The paper that is best known was submitted to SOSP, which in 1987 accepted 19 papers out of 100 submissions. Of those 19 papers, six were selected for publication in ACM TOCS, the premier journal in the field, after another round of refereeing. Our paper, which describes the distributed systems aspects of Emerald paper and its implementation [42], was one of those six, and has been cited 390 times. A later paper [37] that summarizes the non-distributed aspects of the Emerald language design is cited 58 times; even an unpublished technical report on the Emerald type system [60] has been cited 19 times. (All these citation counts were obtained from citeseer.ist.psu.edu). I have selected the TOCS paper for the scrutiny of the tenure committee because it is the most widely cited.

3 Research in Industry: 1987–1994

In 1986 I left the University of Washington and moved to Digital Equipment Corporation, where I created and managed the Distributed Systems Advanced Development group. Later, after the founding of the Cambridge Research Laboratory, I became a member of the research staff at that laboratory. My initial goal at Digital was to take the lessons of the Emerald work and to adapt them to a commercial context, so that they would be available to a larger group of users. Since a special-purpose distributed programming language did not seem to be commercially viable, we instead implemented our system, called Hermes, through extensions to Modula 2 using a pre-processor. One of the major challenges of the environment at Digital was scale; at that time the company's internal network was the largest private network in the world, with on the order of 60 000 hosts. The algorithms that I developed for locating objects in such a network became the subject of two papers [38,39] and a patent [66]. The first paper [38] was published at the leading distributed systems conference, ICDCS, where its impact was such that I was invited to prepare an extended version [39] to inaugurate a new IEEE Journal, IEEE TPDS

Much of my time at Digital was spent as an internal consultant to various of the Massachusetts-based product groups, and also to Digital's representatives to the Object Management Group, which was in the process of standardizing what became CORBA, a restricted subset of the functionality that we had implemented in Emerald. However, in addition to Hermes, two other research topics from this period are worthy of note: Gaggles and Transactions.

Gaggles are an attempt to provide language support for replication in an object-oriented way, the goal being to make it easier to program with replicated objects. Failure is one of the fundamental challenges of distributed computing; replication is vitally important because replication (in time and in space) is the tool that we use to overcome failure. Many techniques have been devised for replicating resources, and it is generally easy to implement the replicas as distributed objects. However, the resulting replicated resource is not itself an object. Moreover, any attempt to represent it as an object (for example, by storing references to the multiple objects that make up the replicated resource in a set) re-introduces a single point of failure. The paper *Encapsulating Plurality* [31] attempts to resolve this difficulty by providing system and language support for *Gaggles*, which are object references that name multiple objects. The paper describes their semantics and an efficient distributed implementation, subject to the imposition of a monotonicity constraint, and was published in the leading European venue for work on object-orientation, ECOOP, which is even

more selective than OOPSLA; in 1993 it accepted 23 papers out of 146 (less than 16 per cent). If I were permitted to select a fourth influential paper for the tenure committee, this would be it.

My work on Transactions started with a 1990 workshop paper [36]. While transactions simplified certain kinds of distributed programming problem enormously, their application was limited to situations in which all of the four so-called "ACID properties" were required. It seemed to me that it would be useful if these properties could be made available as separate operating system services, so that programmers could choose the properties that they really needed, and languages could give them uniform, and if possible compositional, semantics. The workshop paper argued that this was a desirable goal, and took some tentative steps towards attacking it, but did not achieve it. From time to time I returned to the task started in this paper, but it was not until 2001 that, in collaboration with colleagues from EPFL, we gained traction on the problem; the result in summarized in publication 8. Interestingly, the proof of soundness of the formalization presented in this paper relies on an automated logic system, and would have been infeasible in 1990.

Throughout this period, I was consistently sought out as a programme committee member for SOSP (1989 and 1991) and the SIGOPS European workshop (1992), and as a panel member at many workshops and conferences, including the SIGOPS European workshop in 1992 and the ECOOP workshop on Object-based distributed programming in 1993. In 1991 I was asked to serve as general chair of SOSP '93, a signal honor in the systems community. At the same time, I remained active in the languages community; I was a member of the 1990 SIGPLAN conference programme committee, was invited to give a tutorial on types at OOPSLA '92, and served on the programme committee of what became the First Workshop on Foundations on Object-Oriented Languages (FOOL).

4 Oregon: 1994–2004

In 1994 I left Digital to take up the position of Head of the department of Computer Science & Engineering at the Oregon Graduate Institute (OGI). This was a challenging period for OGI, and a demanding one for the head of the largest department in the Institute; there was little time left for my own research agenda. I was able to engage in some productive collaborations with colleagues at OGI [24, 26, 28] and Glasgow [27]. Sometimes there was even research to be done in apparently administrative tasks: publication 23 was a product of the collaboration with PSU, OSU and U of O to design the state-wide Master of Software Engineering program. I also continued to be asked to support the research community, and although it was necessary to restrict this activity because of my commitments to the institution, I was able to chair the IEEE International Workshop on Object-Orientation in Operating Systems in 1996.

Inevitably, my personal research contributions during this period were more limited. Publication 29 foresaw the emergence of "the web" as a computing platform in its own right. In 1998 I was invited to give the banquet address at the European Conference on Object-Oriented Programming (ECOOP '98). I took the opportunity to offer my perspective on the history and the future of the field. This was not quite what the audience expected (I was chastised for being insufficiently funny), but the audience nevertheless found the talk stimulating and I was invited to prepare a paper summarizing the talk for publication in the following year's ECOOP proceedings [22].

In January 2000 I stepped down as department head and returned to research. Since that time I have developed two distinct but related research threads: Infopipes and Multiview.

Infopipes is a collaboration with Jonathan Walpole that grew out of the realization that while streaming media make up an increasing proportion of Internet traffic, streaming applications are not well-supported by the conventional abstractions of distributed computing. In particular, RPC and its object-oriented analogue Remote Method Invocation (RMI) are communication primitives that offer reliability but say nothing at all about latency and bandwidth. This is not an accident but a design decision: reliability is obtained using retransmission, so attempting to provide reliability inevitably means sacrificing latency and consuming additional bandwidth. For media streaming applications, this is the wrong tradeoff: media applications *require* control of latency and bandwidth, and must trade off data integrity if resources are inadequate.

The goal of my Infopipes work is to provide programmers working on streaming applications with tools that are like RPC, in that they hide the boring and mechanical details of communication, but that are unlike RPC in that they give the programmer control of the communication channel rather than hiding it. Moreover, the controls that are given to the application are calibrated in application terms, rather than in the terms of the underlying implementation. For example, the bandwidth of a video Infopipe is measure in frames per second, not bits per second. This work is ongoing; it was initially funded by DARPA and is now funded by an NSF ITR grant. In addition to this funding, the concrete output of this work includes publications 11, 12, 16–19 and 21. Publications 11 (Software P&E) and 12 (Multimedia Systems Journal) were solicited by the editors of the respective journals because of the perceived importance of this work; I have selected the latter as my second representative paper for the tenure committee.

The Multiview project [6] is a collaboration with Mark Jones, Stéphane Ducasse and Oscar Nierstrasz that seeks to do nothing less than change the way that programmers think about their craft. My experience with Smalltalk, with which I first became familiar through my teaching and which later formed the vehicle for my exploration of Infopipes, has convinced me that the programming environment has the potential to become more important than the programming language and that the support provided (or not provided) by the environment has an enormous influence on the perceived difficulty of the programming task.

Multiview takes the position that viewing a program as a linear sequence of symbols on paper or on a display is outmoded and unnecessarily restrictive. Instead, programs should be regarded as complex multi-dimensional artifacts on which linear text provides but one possible view. Freeing ourselves from these restrictions is very difficult, but the potential benefits are enormous. Multiple views make it easier to understand complex programs, and provide a unifying framework for many common program refactorings. In addition, Multiple views provide a solution for language designers trying to choose between competing alternatives: provide the advantages of both, but in different views. Multiview is an ongoing project, which has been funded by the NFS, by OTI, by ETIC and by IBM.

Refactoring is classically thought of as the act of transforming one program into a second, equivalent, program. The point of departure for Multiview is the realization that each refactoring can also be thought of as defining an equivalence relation on programs. Thus, a refactoring partitions the space of all programs into equivalence classes. If we shift our attention from the

individual elements of these equivalence classes to the classes themselves, we have raised the level of abstraction of the programming process.

How are these equivalence classes to be presented to the programmer? How can we focus on the essence of the equivalence class — the common behavior of the witness programs — and ignore the superficial differences? We believe that the answer is to provide *multiple views* on the equivalence class, that is, to represent it by different witness programs depending on the current needs of the programmer, and to make it possible to switch almost effortlessly from one view to another.

The so-called "expression problem" provides a nice illustration of this idea. Consider an abstract syntax tree representing an expression in a compiler. Certain programming tasks, such as adding new operations on the tree, are simple in a language that defines operations by cases over data structures, but are more difficult in a language that puts the data in objects, each of which encapsulates its own operations. Other programming tasks, such as extending the compiler by adding new kinds of node to the tree, are simple in an object-based language, where all that is necessary is to define a new class of object, but are more difficult in a language that defines its operations by cases over data structures, since all of the existing definitions must be extended.

The idea behind Multiview is that it allows the programmer to switch from one view to the other and back again, without loosing context, so that the programmer can always be working in the view that makes the current task easy. Moving from this idea to a useful programming tool requires not only a programming environment that supports seamless view changes, but also a programming language that is expressive enough to encompass both views. This is not always available. For example, ML allows the programmer to define operations by cases, but does not provide a way of encapsulating data and operations; Java does the opposite: it provides classes, which encapsulating data and operations, but does not allow these operations to be defined by cases.

Traits for Smalltalk are a language extension that enable classes to be viewed in multiple ways [2–5, 7, 9–12] Traits were developed in collaboration with Nathanael Schärli, a research student at the University of Bern and his professors, Stéphane Ducasse and Oscar Nierstrasz. It is not necessary to share the Multiview vision to see in Traits a simple addition to the Smalltalk language that solves a long-standing problem: how to share code between classes when inheritance is not up to the job.

The response of the community to Traits has been gratifying. The open-source Squeak Smalltalk community has expressed significant interest in incorporating traits into the base of Squeak, and Cincom, the leading provider of commercial Smalltalk (VisualWorks) has also expressed interest in incorporating traits into their product. At OOPSLA '02, Nathanael and I spent the whole of one day in the exhibit hall giving demonstrations of the trait implementation to interested attendees. At OOPSLA '03, where the results of our first substantial experiment using traits to refactor a substantial Smalltalk class hierarchy were reported [9], a special session was dedicated to Smalltalk, and we were given extra time to present our results and demonstrate the traits programming tools. That this should occur at OOPSLA, which in 2003 accepted only 26 out of 147 submissions (less than 18 per cent), is a measure of the importance attached to this work by the community, and I have consequently selected this as the third paper for review by the tenure committee. There has been significant interest in incorporating traits into other languages;

in addition to our own work on Traits for Java, Reppy and Fisher have proposed static typing rules for Java traits, and Luigi Liquori has proposed a Trait-based extension to Featherweight Java. Microsoft is funding a project in Bern to introduce traits into C♯, and Odersky's recently launched Scala language incorporates a form of traits. Charles Smith, a student of Sophia Drossopoulou at Imperial College, London, has just completed a Master's thesis on Traits. For a language feature that is barely two years old, this is a substantial amount of interest.

5 Summary

Have I achieved my research goal? Is programming in general and distributed programming in particular easier now than it was in 1977? How much of that progress is due to my research?

These questions cannot be answered definitively. It is clear that object-oriented message send between remote objects has become one of the standard mechanisms for encapsulating communication in distributed systems. This technology was pioneered in Eden [51] and Emerald [42]. Will Infopipes become as dominant a technology for rate-sensitive applications? Will traits become a standard feature in the next generation of object-oriented languages? In another 10 years we will know the answers to these questions, but at present we can only look at the opinion of the community, which has rated both very positively, and continues to seek my technical participation in the future of the field. In 2002 I was invited to deliver the keynote address at FOOL, and this year I was asked by the board of AITO to serve as chair of the ECOOP programme committee, arguably one of the highest technical honours in the O-O field.