What can Programming Language Research Learn from the Philosophy of Science?

Tomas Petricek[1]

Abstract. As a recent discipline, computer science, and programming language research in particular, have so far eluded the eyes of philosophers of science. However, we can gain interesting insights by looking at classical works in philosophy of science and reconsidering their meaning from the perspective of programming language research.

This is exactly what I attempt to do in this essay – I will go through the established theories of science and look what they can say about programming language research. Then I suggest how we can improve our scientific practice in the light of these observations.

First, I discuss how understanding the research programme is important for evaluating scientific contributions. Second, I argue that overemphasis on precise, mathematical models in early stage of research may limit the creativity. Thirdly, I propose how to design stand-alone (theory-independent) experiments in programming language research and how this can help to integrate the vast amount of knowledge gathered by software practitioners.

1 INTRODUCTION

Programming language (PL) research is, no doubt, an important part of computer science. It raises many intriguing philosophical questions, ranging from the nature of programming languages to the scientific practice of programming language researchers.

This essay focuses on the latter question – what is the structure of programming language research, how is it done and how should it be done? Despite a number of pioneering works[2] in the philosophy of computer science, this is largely an unexplored domain.

This essay does not aim to present a comprehensive view. Instead I choose a number of classic works in the philosophy of science and look how their observations apply to programming language research.

The basis for many of the classic works used as an inspiration in this essay has been summarized by Chalmers as follows:

> *The undoubted success of physics over the last three hundred years (...) is to be attributed to the application of (...) 'the scientific method'. Therefore, if [other disciplines] are to emulate the success of physics then that is to be achieved by [understanding and applying this method]*[3].

This can be easily viewed as a too narrow approach. Following the *scientific method* may not be the only (or the best) approach. We can equally learn from the *mathematical method*, the *artistic method* or from *social sciences*[4]. Similarly, physic is not the only successful discipline. However, this is the view adopted by the classical works in philosophy of science that I take as an inspiration and that I follow in this text.

My main goal is to demonstrate that taking the philosophical perspective on programming language research is a worthwhile effort that can lead to interesting ideas. With no doubt, future work needs to take broader and more comprehensive view. In this short essay, I look at the following questions:

- In what way is programming language research a *science*? I look at Popperian falsificationism (§2.1), experimentalist view (§2.2) and Feyerabend's anarchistic perspective (§2.3).
- What is the structure of programming language research? Are there different scientific paradigms (§3.1) or competing research programmes (§3.2)?
- How can we use the above ideas to better judge contributions (§4.1), support creative research (§4.2) and produce more reusable experiments or software artifacts (§4.3 and §4.4).

This essay is intentionally written in a subjective and perhaps provocative style. I believe that the questions posed in the first two sections (and discussions they can trigger) are equally important as my answers suggested in the last section.

2 PL RESEARCH AS A SCIENCE

A commonsense understanding of science is that science starts with an unprejudiced observation of reality, infers facts from these observations and uses sound reasoning to derive scientific laws. For a moment, I shall ignore the numerous problems with this view and look what is the corresponding practice in programming language research.

Do programming language researchers observe the reality? In some cases, they do – they study programs written by the industrial engineers or programs (artifacts) created as a result of earlier research. However, just observation is not enough to obtain *relevant facts* and so programming language researchers perform experiments – they write compilers for their languages and use them to write sample programs; they implement novel algorithms and test their performance. Broadly speaking, such experiments can be classified in two categories. The first kind is constructed to *confirm a specific theory* (e.g. the performance of new garbage collection algorithm in practice). Second kind is constructed for

[1] Computer Laboratory, University of Cambridge, CB30FD, UK
Email: tomas.petricek@cl.cam.ac.uk

[2] Most prominently Turner [23] and a number of works in the ethics of computer science and philosophy of artificial intelligence.

[3] Chalmers [2], *xx*

[4] For example, Meyerovich and Rabkin [16] use sociologically grounded approach to study language adoption

the *experiment itself* (e.g. using a new language to develop a system is an interesting case study on its own).

What kind of facts do we derive from the observations? Some observations, such as performance measurements, yield unquestionable facts in the form of statistics. However, the facts derived from programming language experiments are less clear. Small-scale examples often included in publications are sufficient to demonstrate that a language is capable of expressing certain abstractions or preventing certain bugs. However, they often do not present a realistic study of how a language would be used in practice. This is a separate challenge that has been explored in the early days of programming language design by Sime et al. [21], but is again becoming an interesting topic – see, for example the workshop proceedings edited by Murphy-Hill et al. [17].

Finally, what kind of laws do programming language researchers arrive at? PL research often consists of building of simplified models (*semantics*) of languages and proofs of their properties. Much can be said about the nature and the meaning of proofs[5], but there is a more important hidden assumption – we believe that proving facts about simplified model tells us important information about the applicability of the language in the "real world." Perhaps a more interesting claim that is often implicit[6] is that a language can capture some common mental model that its users use when thinking about problems.

2.1 Falsificationism

I suggested several laws (or results) that programming language researchers may claim, but I am not surprised if the readers find the examples unconvincing. Indeed, it is difficult to find programming language research that makes explicit claims in the form of traditional scientific laws.

What would a meaningful scientific claim look like? The most well-known answer is provided by Popper's falsificationism:

> *I shall certainly admit a system as (...) scientific only if it is capable of being tested by experience. These considerations suggest that not the verifiability but the falsifiability of a system is to be taken as a criterion of demarcation[7].*

It is not necessary to prove that the claim is true, but it must be possible to refute it by an empirical test. Finding programming language claims that do not fail this criteria is difficult. A claim about mathematical model cannot be refuted by experience; a claim about language usability passes, but it cannot resist refutation for a long time – it will hardly hold universally for all users.

However, it is worth noting that Popper does not devise the above test as a test for worthwhile human activity, it is just a test (or *demarcation*) for empirical sciences:

> *The problem of finding a criterion which would enable us to distinguish between the empirical sciences on the one hand, and mathematics and logic as well as 'metaphysical' systems on the other, I call the problem of demarcation[8].*

According to Popper's theory, programming language research does not qualify as *empirical science*. Some aspects of programming language research would clearly belong to the category of mathematics and logic, but a large proportion of work falls in the other category[9]. Let us now consider an alternative treatment.

2.2 The new experimentalism

Rather than defining what claim is scientific, we can sidestep the problem, focus on another aspect of science and take experiments as the basis of our philosophy of programming language research.

As already discussed, PL researchers do make experiments. They implement compilers or interpreters, use them to develop sample applications or measure performance of systems. But do such experiments make sense only as part of a theory, or do they have a value on their own?

Indeed, many scientific experiments are theory-dependent. For example, experiments designed to confirm the existence of an aether in 19th century[10] became irrelevant once physics abandoned the idea of an aether. Similarly, an implementation of a compiler for a programming language is only relevant in the light of the given programming language[11]. A group of philosophers[12] sometimes called *new experimentalists* believe that theory-independent experiments can form a foundation of the scientific method:

> *According to its proponents, experiment can (...) have a "life of its own" independent of a large-scale theory. It is argued that experimentalists have a range of practical strategies for establishing the reality of experimental effects without needing recourse to large-scale theory[13].*

Another interesting aspect of new experimentalism and its focus on experiments is that it provides a notion of scientific progress. Accumulated experimental knowledge remains valuable even when new theories appear. The same theory-independence also means that radically different theories can be compared, for example, by looking which established experiments they explain.

As a programming language researcher who wants to subscribe to the "new experimentalist" approach I need to ask: "How to produce theory-independent experiments in PL?" In physics, an example of such experiment is Faraday's motor (the first electrical motor to be built, also called *homopolar*)[14] – it is an easy to build device that is not fallible (it usually works) and has obvious effects (rotation) that a theory needs to explain, be it the theory of electromagnetism or other theory that we may devise.

[5] First section in Gold and Simons [12] discusses relevant questions: Does the formalization of proofs increases the reliability? What is the purpose of proofs? It is not just convincing the reader about the truth – it is also an explanation of the problem, exploration yielding new insights and justification of the definitions (in our case, programming language design).

[6] *Implicit* possibly because the predominant paradigm dictates that such claim is "unscientific" when based just on the author's introspection.

[7] Popper [18], 18

[8] Ibid., 11

[9] That said, falsifiability can provide interesting insights from the programming language perspective (Forster 2008), just not as an overall scientific theory.

[10] Chalmers [2], 36

[11] Note that I am, by no means, suggesting that a compiler for a newly designed programming language is not a useful artifact. It is valuable in that it allows further experimentation. However, it is on its own not a theory (language) independent artifact that yields new insights not related to the particular programming language (or paradigm) studied.

[12] Pioneering work in this direction is Hacking [7]

[13] Chalmers [2], 194, quoting Hacking [7]

[14] Ibid., 196

I return to the topic of theory-independent experiments in programming language research later (§4.3). Briefly – I propose that a medium-scale practical case study is an experiment showing that certain problem can be solved with such and such properties. Furthermore, case study is an artifact that both PL researchers and practitioners can understand and learn from.

2.3 Against method

Programming language research does not seem to easily fit the commonsense view of science or the falsificationism approach. While experimentalism is an attractive alternative, it focuses only on one particular aspect of science. Is there a more appropriate view of science that better fits programming language research?

To avoid future disappointment, even traditional sciences do not easily fit structures described by philosophers of science. This led Paul Feyerabend to formulate his anarchistic theory:

> *To those who look at the rich material provided by history (...) it will become clear that there is only one principle that can be defended under* all *circumstances and in all stages of human development. It is the principle:* anything goes[15].

Feyerabend says that scientific ideas are developed in much less organized manner than what its image suggests. He gives examples from history where newly proposed (later successful) theories contradict (the current understanding of) experimental results[16] and rely on ad-hoc approximations[17]. For example, theories developed by Galileo were in direct conflict with scientifically accepted "facts" of his time. In addition to intellectual reasons, Galileo employed propaganda[18] to change such established natural interpretations.

The brief example illustrates the point that Feyerabend makes in a more elaborate way. The history of science shows that there is no universal scientific method and *"science is an essentially anarchic enterprise"*. However, this is not a bad thing:

> *[T]heoretical anarchism is more humanitarian and is more likely to encourage progress than its law-and-order alternatives.*[19]

Such view of science does not provide any guidelines for distinguishing "good science" and "bad science". This is an interesting point for programming language researchers. Many languages used in practice do not qualify as "good science" according to commonsense PL research perspective. Yet, they are popular and widely used. The anarchistic perspective offers hints on how to take such languages into consideration and learn from them, even though they do not originate from the scientific method.

This does not mean that we should study everything ever created. Feyerabend comments his selection procedure as follows:

> *I make my selection in a highly individual and idiosyncratic way. (...) Science needs people who are adaptable and inventive, not rigid imitators of 'established' behavioural patterns*[20].

Many philosophers science view Feyerabend's anarchistic perspective as too radical. Even for programming languages, where choice is often very subjective, it is difficult to imagine how a fully subjective approach could be employed.

Chalmers [2] attempts to find a middle ground. He argues that there are some scientific standards, but these can change (which leaves enough room for the "anything goes" method). I take similar position when I argue that early work needs to be less mathematically precise than work in more developed domains (§4.2).

3 STRUCTURE OF PL RESEARCH

Kuhn and Lakatos are two influential philosophers of science who attempt to capture the structure of scientific development by looking at the history of science and propose theories that capture well-known examples of scientific practice (such as Galilean and Newtonian revolutions in physics). Unlike the works discussed in the previous section, they do not dictate how science should be done. They merely attempt to describe the historic reality.

Applying this methodology to the history of programming languages is an interesting problem, but one that I leave to future work. However, such treatment of science also explores assumptions that remain hidden during regular scientific practice, but influence how science is done (e.g. which established "facts" can be questioned & revisited; in what circumstances and how).

3.1 Scientific revolutions

According to Kuhn [13], science (after initial pre-scientific phase) proceeds in cycles where a period of *normal science* is followed by a crisis and a *scientific revolution* that leads to a new period of normal science. A period of normal science is governed by predominant scientific paradigm:

> *A paradigm is made up of the general theoretical assumptions, laws and the techniques for their application that the members of a particular scientific community adopt*[21].

The paradigm dominates and entire field (or a subfield) and its existence is what makes normal scientific work possible:

> *When the individual scientist can take a paradigm for granted, he needs no longer, in his major works, attempt to build his field anew, starting from first principles and justifying the use of each concept introduced*[22].

The paradigm is what each aspiring scientist learns during his or her preparation. The assumptions of the paradigm are so ubiquitous that *"normal scientist will be unaware of and unable to articulate the precise nature of the paradigm"*[23].

The only moment when scientists become aware of the assumptions dictated by the paradigm is during the period of *crisis*. That is, when the paradigm is found insufficient for solving problems (puzzles) within the normal science. In that case, the predominant paradigm is replaced with another:

[15] Feyerabend [5], 12
[16] Ibid., 13
[17] Ibid., 43
[18] Ibid., 61
[19] Ibid., 1
[20] Ibid., 163
[21] Chalmers [2], 108
[22] Kuhn [13], 19
[23] Chalmers [2], 112

[S]cientific revolutions (...) [are] non-cumulative developmental episodes in which an older paradigm is replaced in whole or in part by an incompatible new one[24].

Programming language research as a whole is likely too young for identifying such paradigm shifts (it would be a mistake to view programming language paradigms as Kuhnian paradigms). However, we can try to uncover the background assumptions commonplace in the PL research today.

For a programming language researcher, this is, indeed, a difficult task! I believe that one such assumption is the reliance on simplified mathematical models – everyone agrees that such models provide useful insights. I do not want to doubt this, but the amount of trust in models is surprising when the aim is often to produce much larger industrial-scale implementations[25].

Aside from common assumptions, the paradigm also provides techniques that are employed when facing a problem. An example of such technique from programming language design might be the approach to rule out bugs using a *type system*. The paradigm also dictates what is required of such type system – for example the need for soundness. Yet, this requirement of the traditional research paradigm has been ignored in several recent programming languages originated in the industry such as Dart[26].

3.2 Research programmes

Another attempt to explain the structure of science has been made by Lakatos [14]. He looks how has falsification been used in science in the past and notes that the failure of a theory can be ascribed to different aspects of the theory – there is no single assumption to blame. Lakatos also notes that not all assumptions are equal. Scientists can always protect a theory they believe by ascribing failures to less fundamental assumptions.

This is the basis for Lakatos's theory of *research programmes*. Similarly to paradigms, research programmes specify the background assumptions. Unlike with paradigms, science consists of multiple competing research programmes formed by groups of scientists. A research programme develops as follows:

Scientists can seek to solve problems by modifying the more peripheral assumptions (...). [T]hey will be contributing to the development of the same research program however different their attempts (...). Lakatos referred to the fundamental principles as the hard core[27].

The *hard core* is a defining characteristic of a research programme. It is augmented with a *protective belt* of auxiliary assumptions that can be freely modified. The assumptions forming the hard core are essentially unfalsifiable and all failures of theories are attributed to the protective belt.

We can take a purely functional programming as an example of a research programme in PL research. The hard core is formed by concepts such as immutability, pure functions and the lack of side-effects. An experimental failure (e.g. difficulty in implementing an efficient algorithm) is attributed to the auxiliary assumptions, such as an insufficient optimization in the compiler, but not to the hard core assumptions like immutability.

According to Feyerabend, this methodology is so lax that it can accommodate almost everything[28]. Lakatos himself claims that there is no instant rationality in science and he does not treat his philosophy as an advice to scientists[29]. The structure of research programmes can be fully reconstructed only in hindsight.

Lakatos's philosophy still provides some structure. It might not rule out "bad science"[30], but it provides a way to navigate through the complex web of PL research that is otherwise ruled by the "anything goes" methodology. I suggest that acknowledging the hard core assumptions that a programming language researcher subscribes to can improve the practice of our field (§4.1).

3.3 Beyond philosophy of science

So far, we looked at programming language research through the scientific perspective that has been inspired by physics. However, there are other successful disciplines that have much to say about the structure of human enterprises. I briefly mention mathematics and social sciences. Devlin answers the question about the nature of mathematics as follows:

[A] definition of mathematics (...) on which most mathematicians now agree, and which captured the broad and increasing range of different branches of the subject [is]: mathematics is the science of patterns[31].

This sounds very familiar to programming language researchers. Such key concepts as abstraction[32] are essentially patterns and PL design is about finding better ways to capture such patterns. The philosophy of mathematics can shed light on other aspects of programming language research, including the nature of models and their relation to reality.

Another similarity between mathematics and programming language research is that they both construct (and affect) the structures that they study. For Lakatos, this is a reason why theories of science derived from physics may not be applicable to other disciplines – although he speaks of social sciences and, more specifically, economics:

[F]or example, economic theories can affect the way in which individuals operate in the market place, so that a change in theory can bring about a change in the economic system being studied[33].

Aside from being self-referential, economics is similar to programming language research in its emphasis on mathematical models. In his recent book, Sedlacek rethinks economics from a broader perspective that does not focus just on this aspect, but includes long history of myths and religions:

[24] Kuhn [13], 92

[25] Interestingly, programming language research is not the only discipline that glorifies mathematical models. The same is the case for main-stream economics. This has been noted by Sedlacek: "We economists are frequently not even really aware of what we say with our models. This is caused by devoting more attention to (mathematical) methods than to the problems these models are being applied to." (Sedlacek [19], 288)

[26] See Brandt [1] for explanation and the motivation

[27] Chalmers [2], 131

[28] Ibid., 154

[29] Ibid., 144

[30] Lakatos distinguishes between progressive and degenerating research programmes, but he does not say that scientists should abandon the latter ones, because new research can always bring them back to life.

[31] Devlin [4], 293.

[32] Turner and Eden [23] discus abstraction from the philosophical view

[33] Chalmers [2], 47

It would be foolish to assume that economic inquiry began with the scientific age. At first, myths and religions explained the world to people who ask basically similar questions as we do today[34].

While learning from myths and religion may be a bit far-fetched for programming language design, there is another source of knowledge that is often ignored and can provide enormous value. I am, of course, speaking of the skills and practices of a broad programmer community.

While programming language researchers often aim to solve the problems that are faced by the IT community, we tend to dismiss "commonsense wisdom" of software developers as unscientific. We are throwing out the baby with the bath water and I consider how to remedy this problem in the next section (§4.4).

4 LEARNING FROM PHILOSOPHIES

The previous two sections were an exploration of – following Feyerabend's criteria for selection – subjectively chosen works in philosophy of science and an attempt to demonstrate their relevance to programming language research. In this section I go further and make three concrete suggestions how to improve the scientific practice of PL research.

First, Feyerabend's anarchic presentation of history follows the slogan "anything goes". Even if we ignore Feyerabend's humanistic motivations[35], his historic account shows that this is how science proceeds. We need to accept this and make our practice more flexible to support the development of competing theories.

Second, such flexibility (or plurality) in programming language research should accommodate theories in their early stages. According to both Kuhn and Feyerabend, theories start imprecise and only develop fully formal methods at later stage.

Third, PL research develops competing theories (programming languages) that are difficult to compare. I believe that Hacking's new experimentalism might be a pathway towards the solution. By focusing on theory-independent experiments, we can compare different theories, but also integrate "non-scientific" knowledge developed by software practitioners.

4.1 Anything goes: A case for plurality

Feyerabend's slogan "anything goes" should not be interpreted as a license to treat anything as science. It means that there is no universal scientific method, but there are still scientific standards:

I argue that all rules have their limits and that there is no comprehensive 'rationality', I do not argue that we should proceed without rules and standards[36].

While we cannot accommodate *all* possible standards (we cannot know what the standards are until we look at science in hindsight), we can certainly identify several different standard in programming language research. Some work is focused on theory (with proofs) while other emphasizes practical implementation (with performance measurements). However, the classification can be more fine-grained.

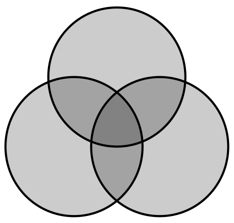

Figure 1. Overlapping cores of competing research programmes

One language feature or PL theory is approached with different interpretations and goals. This is nicely captured by Lakatos's research programmes. Depending on the programme, researchers will work by modifying different auxiliary assumptions.

It is important to understand that different work follows from different *hard core* assumptions. By respecting this difference and making it more explicit (to a certain possible extent), we can avoid judging contributions of a research using incompatible criteria.

Such incompatibility is an inherent part of science. Feyerabend argues that the *consistency condition* (which requires that new hypothesis agrees with accepted theories) is overly restrictive:

[T]he methodological unit to which we must refer [is] a whole set of partly overlapping, factually adequate, but mutually inconsistent theories[37].

In other words, judging new work from the perspective of an existing (incompatible) theory may rule out result that is important from the view of a different theory. While I do not advocate fully subjective approach (as Feyerabend does), I argue that we should try to judge research work from the *right* perspective.

Furthermore, if we evaluate a hypothesis that subscribes to a certain theory from the perspective of *multiple different* theories (which can easily occur in a typical review process), we may find it inacceptable if it modifies the hard core of *any* of the alternate theories. This is demonstrated in Figure 1 – the hard cores of three theories overlap. Each of them would accept hypotheses that fall outside its own hard core. However, hypotheses that are acceptable to all of them can only modify auxiliary assumptions that fall outside of the union of the three research programmes – and so researchers need to be much more conservative in their contributions than their own research programme requires.

The historical evidence discussed by Feyerabend shows that hypotheses inconsistent with established theories (and even generally accepted "facts") can lead to scientific progress. It might not be possible to see which inconsistent theories are worthwhile in advance, but we should, at least, better accommodate the plurality of multiple competing theories.

If we openly allow multiple incompatible theories in our practice, we can also be more honest about our approach. We do not need to conceal the fact that – as put by Feyerabend – *"science is much more 'sloppy' than its methodological image"*[38].

Feyerabend also demonstrates that new theories often take step back and do not necessarily have increased content (they do not add new results). Instead they define new problems or give a new perspective. We can easily find historical evidence from the

[34] Sedlacek [19], 4

[35] According to Feyerabend, his anarchistic account of science "increases the freedom of scientists by removing them from methodological constraints and, more generally, leaves individuals freedom to choose between science and other forms of knowledge (Chalmers [2], 156).

[36] Feyerabend [5], 242

[37] Ibid., 20

[38] Ibid., 160

programming language field where it would have been unwise to reject a novel theory early just because it suffered from problems solved by the established theory.

The example I have in mind is purely functional programming. When first developed, purely functional languages did not have a good way for dealing with I/O and it took time until this problem was solved with linear types and later monads. We needed to *"wait and ignore large masses of critical observations"*[39] until it has been supplemented by the necessary auxiliary techniques.

Aside from ignoring limitations of a new theory, we should also accept the fact that new theories are initially less precise:

> *A new period in the history of science commences with a backward movement that returns us to an earlier stage where theories are more vague and have smaller empirical content*[40].

The increase vagueness of novel theories in early stages leads to the second point of this essay. Is there a room for vagueness and imprecision in programming language research?

4.2 Early stages: A case for inexactness

It is widely accepted that programming languages should be based on solid foundations and precise mathematics. I do not wish to dispute this – the role of mathematics in guaranteeing safety and robustness of languages is unquestionable.

However, the history of science provides a strong argument for including inexact hypotheses and other works as part of the scientific practice. There is a rare agreement among philosophers of science mentioned in this essay that early phases of science (be it paradigms, research programmers or science in general) are often vague and inexact.

The following quotes by Feyerabend and by Chalmers (when describing the work of Kuhn and Lakatos) illustrate the point:

> *Logically perfect versions (if such versions exist) usually arrive long after imperfect versions have enriched science by their contributions*[41].

> *A case could be made to the effect that the typical history of a concept (...) involves the initial emergence of the concept as a vague idea, followed by its gradual clarification as the theory (...) takes a more precise (...) form*[42].

> *Early work in a research program is portrayed as taking place without heed or in spite of apparent falsifications by observation*[43].

The history also shows that such "early phases" of scientific hypotheses or a research programmes are often surprisingly long. I believe that the same is the case for programming language research and that rejecting imperfect or unmathematical versions of research is not beneficial for the field.

So, what form of early or vague research might be interesting? Here are some examples we can learn from. Kuhn suggests that Galileo's early efforts *"involved thought experiments, analogies and illustrative metaphors rather than detailed experimentation"*[44]. According to Lakatos, an important aspect of early stages of research programmes are confirmations – cases where the programme succeeds at predicting phenomena (or explaining an important problem), despite apparent falsification of other aspects of the programme.

Similarly, I argue that analogies, illustrative metaphors and thought experiments are equally worthwhile for programming language research. Case studies that show the applicability of a language in an important domain, as advocated later (§4.3), can fill the role of early confirmations.

Another motivation for adopting more lax rules in early stages of research programmes is that the focus on precise mathematics and clarity changes the perspective and can draw the attention away from the original motivations. In other words, it means that different problems matter. Feyerabend says the following about the undesirable consequences of an early clarity requirements:

> *The course of investigation is deflected into the narrow channels of things already understood and the possibility of fundamental conceptual discovery (...) is considerably reduced*[45].

I very much agree with this quote and believe that programming language research needs to start with the focus on fundamental (non-technical) questions and then gradually evolve – including the clarification and the development of mathematical theory.

Interestingly, very similar words have been recently said about other disciplines. Sedlacek writes the following about mathematical models in economics:

> *It appears to me that we have given lawyers and mathematicians too large a role at the expense of poets and philosophers. We have exchanged too much wisdom for exactness (...)*[46].

Even more interestingly, the call for freer and more liberal reasoning has also been made by mathematicians themselves:

> *Too strong an emphasis on proof may thus be more of an impediment than an aid to the development of new mathematical theories. To become more efficient (...) mathematics should follow the lead of physics and permit freer use of intuitive methods of thinking*[47].

The form of freer and more intuitive methods of thinking in programming language research will certainly vary. One possible form that I wish to discuss in more detail is the form of a case study.

4.3 Experimentalism: A case for case studies

There has been a number of calls recently in computer science[48] as well as in programming language research[49] to increase the focus on experimentation and artifacts. The goal of such movements is to follow other sciences and enable programming language researchers to learn from empirical observations. Furthermore, the publication of artifacts (reproducible experiments)

[39] Ibid., 112
[40] Ibid., 112
[41] Feyerabend [5], 8
[42] Chalmers [2], 106
[43] Ibid., 135
[44] Ibid., 106
[45] Feyerabend [5], 200
[46] Sedlacek [19], 321
[47] Detlefsen [3], 9 discussing Jaffe, Quinn [10]
[48] For example, see Feitelson [20]
[49] Hauswirth [8]

should also enable further research. Quoting from the call for artifacts at OOPSLA 2013:

> *The high level goal of the Artifact Evaluation (AE) process is to empower others to build upon the contributions of a paper*[50].

According to the call, artifacts should be consistent with the presented theory, as complete as possible, well documented and easy to reuse. Such artifacts allow confirmation of experimental claims but they are not necessarily a good experiment *per se*:

> *That [experiments] are adequately performed is necessary but not sufficient condition for the acceptability of experimental results. They need also to be relevant and significant*[51].

What does it mean for a software artifact to be relevant and significant? A relevant artifact should be an empirical confirmation of its claims – this is partly the purpose of aforementioned artifacts, but the (confirmable) claims need to be explicitly stated, be it performance or the ability encode some mathematical pattern without cognitive overhead.

There is a difference between an artifact (or an experiment) that is relevant to a single research programme and an artifact (experiment) that is relevant to the programming language research as a whole. I argue that only the latter kind is *significant*.

The new experimentalism, introduced in an earlier section, makes a similar call. The crucial claim is that *"experiment can have a life of its own"* and can be independent of theory (or a research programme). Chalmers presents historical evidence that such experiments are possible and summarizes:

> *The production of controlled experimental effects can be accomplished and appreciated independently of high-level theory*[52].

If we treat programming languages or language features as theories then a theory-independent experiment needs to be an artifact that uses a particular language, but has an observable value regardless of the particular language details. Recall the example of Faraday's motor – it may have been constructed to demonstrate electromagnetic theory, but it has a clear and interesting observable effect (rotation).

Similarly, artifacts in PL research could be systems that solve some non-trivial problem and show that the solution can have certain properties (e.g. is provably correct, closely corresponds to some mental model or has other notable properties that cannot be easily achieved using other languages).

As mentioned earlier, I argue that *case studies* provide a way for constructing such theory-independent artifacts. Let us examine this in light of the following definition of a case study:

> *Case study is an in-depth exploration from multiple perspectives of the complexity and uniqueness of a particular project, (...), program or system in a "real life" context*[53].

The most common artifact provided when discussing a novel language, feature or a tool is an implementation (e.g. a compiler or a library). While this is sufficient *"to empower others to build upon the contributions"*, it does not offer multiple perspectives and "real life" context. These can be added by applying the tool to a number of practical problems (such as development of non-trivial system) and the analysis of such implementations.

Indeed, programming language research often aim to improve the practice in a "real life" context. However, this is difficult to express as a scientific claim and so it often remains implicit or unacknowledged. I believe that we need to accept that PL research is not just mathematics and learn from social sciences. We should accept the need for a more holistic approach that can be employed in case studies.

Another key aspect of the new experimentalism is that theory-independent experiments can be used to compare radically different theories or, in our case, programming languages:

> *Implicit in the new experimentalist's approach is the denial that experimental results are invariably "theory" or "paradigm" dependent to the extent that they cannot (...) adjudicate between theories*[54].

I believe this should also be the case for case studies in PL research. Artifact such as compiler implementation is clearly insufficient for comparison of multiple theories (languages). However, if we had case studies solving related problems (e.g. implementing similar systems) in different languages, we would be able to compare properties of the languages for one particular scenario. This gives us a way to contrast the experience with earlier work – even if the comparison is going to be more subjective than in formal mathematical treatment.

4.4 Practical experience: A case for inclusiveness

Finally, the evaluation of programming languages in a "real life" context also means that such case studies could provide a common language between programming language researchers and practitioners. I already mentioned the importance of the history and experience of practitioners when discussing how myths and history are important for economics.

For Feyerabend, such wider collaboration is a historical fact and it is necessary for science:

> *[A scientist] who wants to understand as many aspects of his theory as possible (...) will adopt pluralistic methodology (...). For the alternatives (...) may be taken from the past as well. As a matter of fact, they may be taken from wherever one is able to find them – from ancient myths and modern prejudices; from the lucubrations of experts and from the fantasies of cranks*[55].

This is even more the case for programming language research – there is hardly any field of science where the collaboration between scientists and non-scientists is more important and so finding a common language is crucial.

I am not the first one to make such call in the field of programming language research. Meyerovich and Rabkin noted that programming languages are often created by people outside of the programming language research community and discuss the importance of communication:

[50] Hauswirth [9]
[51] Chalmers [2], 37
[52] Ibid., 197
[53] Simons [22], 21
[54] Chalmers [2], 205
[55] Feyerabend [5],27

[P]rogramming language community should not only focus on justifying features to programmers. We should focus on better consulting with the wider software development community to see what is relevant and communicating our findings to new language designers, who usually come from outside of our community[56].

To summarize, we need to focus on theory-independent experiments, both to make our research more honest (by acknowledging implicit claim that we improve the practice) and to make it more useful (by providing value to practitioners). There are surely multiple approaches towards this goal, but I propose case studies as a form of experiments with *"life of their own"*.

This focus has a number of benefits. It allows comparison of radically different theories, it allows us to evaluate our work in a wider "real life" context. Finally, it also encourages involvement of people outside of the narrow PL research field. In other words, we should not *"discard the immense treasures of knowledge and wisdom that are contained in the traditions"*[57].

5 CONCLUSIONS

This essay serves two purposes. Firstly, I argue that philosophy of science is a valuable source of ideas and inspirations for programming language research practice. Secondly, I give a concrete (subjective) answer to the question: *"What can programming language research learn from the philosophy of science?"*

I started with an exploration of classic theories known from philosophy of science. I introduced theories that suggest what methodologies should (or should not) be followed including Popper's falsificationism, new experimentalism and Feyerabend's anarchic theory. I also discussed theories that ascribe some structure to history of science – namely Kuhn's scientific revolutions and Lakatos's research programmes.

In the second part of the essay, I made a case for three ways of improving the established methodology of programming language research. I argued for plurality – that is, we should acknowledge the fact that there are multiple research programmes that consider different problems important and have different aims. I argued for inexactness – history shows that early stages of scientific theories and paradigms are inexact. Requiring early precision limits the creativity and may shift attention from crucial problems of the theory. Finally, I argued for case studies as a way to produce theory-independent experiments that make it possible to compare radically different theories and can serve as a common language between researchers and software practitioners.

[56] Meyerovich, Rabkin [16]

REFERENCES

[1] Brandt, E. (2011). Why Dart Types Are Optional and Un-sound. Online at: http://www.dartlang.org/articles/why-dart-types

[2] Chalmers, A. F. (1999). What is this thing called science? Open University Press. ISBN 0335201091.

[3] Detlefsen, M. (2008). Proof: Its nature and significance. In Proof and other Dilemmas (Gold, B., Simons, R. A., eds.) pp291-311. The Mathematical Association of America.

[4] Devlin, K. (2008). What will count as mathematics in 2100? In Proof and other Dilemmas (Gold, B., Simons, R. A., eds.) pp291-311. The Mathematical Association of America.

[5] Feyerabend, P. (2010). Against method. Verso (4th edition). ISBN 1844674428.

[6] Forster, T. (2008). Falsifiability: what Popper got right. Unpublished. http://www.dpmms.cam.ac.uk/~tf/falsifiability.pdf

[7] Hacking, I. (1983). Representing and Intervening: Introductory Topics in the Philosophy of Natural Science. Cambridge University Press. ISBN 0521282462.

[8] Hauswirth, M. et al. (2013a). Experimental Evaluation of Software and Systems in Computer Science: Letter to PC chairs. Online. http://evaluate.inf.usi.ch/letter-to-pc-chairs

[9] Hauswirth, M., Blackburn, S. (2013b). OOPSLA Artifacts: Call for papers. Online: http://splashcon.org/2013/cfp/665

[10] Jaffe, A., Quinn, F. (1993). Theoretical mathematics: Towards a cultural synthesis of mathematics and theoretical physics. Bulletin of the American Mathematical Society 29, pp.1-13

[11] Gabriel, R. P., Sullivan, K. J. (2010). Better science through art. In proceedings of OOPSLA 2010.

[12] Gold, B., Simons, R. A. (2008). Proof and other Dilemmas: Mathematics & Philosophy. The Mathematical Association of America. ISBN 0883855674.

[13] Kuhn, T. S. (1970). The Structure of Scientific Revolutions. The University of Chicago Press (2nd edition). ISBN 0226458040.

[14] Lakatos, I. (1975). Falsification and the Methodology of Scientific Research Programmes in Can Theories be Refuted? Essays on the Duhem-Quine Thesis (ed. Harding, S. G.), pp205-259. ISBN 9789027706300.

[15] Lorenz, K. (1984). Die acht Todsünden der zivilisierten Menschheit. Piper Verlag, Munich.

[16] Meyerovich, L. A., Rabkin, A. S. (2012). Socio-PLT: Principles for programming language adoption. In proceedings of Onward! 2012.

[17] Murphy-Hill, E., Sadowski, C., Markstrum, S., eds. (2012) Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools.

[18] Popper, K. (2005). The Logic of Scientific Discovery. Taylor & Francis eLibrary. ISBN 0-203-99462-0

[19] Sedlacek, T. (2011). Economics of good and evil: the quest for economic meaning from Gilgamesh to Wall Street. Oxford University Press. ISBN 9780199767205.

[20] Feitelson, D. G. (2006). Experimental Computer Science: The Need for a Cultural Change. Unpublished note. Available online: http://www.cs.huji.ac.il/~feit/papers/exp05.pdf

[21] Sime, M. E., Green, T. R. G., Guest, D.J. (1973). Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, Volume 5, Issue 1, January 1973, pp105-113

[22] Simons, H. (2009). Case study research in practice. Sage Publications. ISBN 076196424X.

[23] Turner, R., Eden, A. (2011). The Philosophy of Computer Science. In The Stanford Encyclopedia of Philosophy (Zalta, E. N. eds.). http://plato.stanford.edu/entries/computer-science/

[57] Lorenz [15] as quoted by Feyerabend [5], 131 footnote 16