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

Tomas Petricek
University of Cambridge, United Kingdom
tomas.petricek@cl.cam.ac.uk

Abstract

Compared to physics, biology or chemistry, we can hardly call computer science an established discipline. The research in the design of programming language is even younger. As such, it has so far eluded the eyes of philosophers of science.

Nevertheless, we can still learn a lot by looking at classical works in philosophy of science and reconsidering their meaning from the perspective of programming languages.

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

First, I discuss how understanding the research programme of work is important for evaluating its contributions. Second, I present evidence for the claim that overemphasis on precise, mathematical models in early stage of research may limit the creativity. Third, I propose how to design more long-lasting experiments in programming language research and how this can help to integrate the vast amount of knowledge gathered by software practitioners.

Introduction

Although *computer science* has the word science in its name, understanding the sense in which it is a *science* is not an easy task. Computer scientists do not study the world with the aim to understand how it works and to find laws that govern it.

This is even more the case in programming language research. Rather than studying the world as it is, we construct new theories and artifacts and then study our creations. The theoretical aspect of programming language research can be more easily related to mathematics, while some like to relate the creative aspects of programming language design to art[1].

I will not attempt to answer the question about the nature of computer science in the present essay. Instead, I will follow the approach summarized by Chalmers in the introduction to his book on philosophy of science:

> *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]*[2].

I am not suggesting that following the *scientific method* is the only (or the best) way for programming language research. Indeed, one can equally learn from the *mathematical method*, the *artistic method* or from *social sciences*[3].

Nevertheless, there are many relations between sciences such as physics and programming language (PL) research. Physicists confirm their theories using experiments while PL researchers implement their theories as software artifacts or use their languages to build applications as case studies confirming their theories about languages. Just like in physics, we generalize from numerous of observations or develop unifying theories to capture previously unrelated concepts.

Finally, programming language research and science share a number of structural aspects. We can identify competing *research programmes* that specify what principles are fundamental and what techniques are employed to answer research questions. We can also search for predominant *paradigms* that determine what simplifying assumptions are acceptable and what questions are considered worthwhile.

[1] The essay Gabriel and Sullivan (2010) in an earlier edition of Onward! Essays relates science and art and draws many examples from computer science.

[2] Chalmers (1999), *xx*

[3] Meyerovich and Rabkin (2012) use this approach to study language adoption

Programming language research as a science

A commonsense understanding of science is that science is derived from facts. It starts with an unprejudiced observation of reality, infers facts from these observations and using 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. 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 the *experiment itself* (e.g. using a new language to develop a system is an interesting case study on its own). I will return to the topic of experiments later.

What kind of facts do we derive from the observations? Some observations, such as performance measurements, yield unquestionable facts in forms of numbers. 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 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. (1973), but is again becoming an interesting topic – see, for example, Murphy-Hill et al. (2012).

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[4], 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[5] is that a language can capture some common mental model that its users use when thinking about problems.

Falsificationism

I suggested several kinds of laws that programming language researchers may claim, but I am not surprised if the readers find the examples unconvincing. Indeed, I believe that it is difficult to find programming language research that makes claims or states laws in the traditional scientific sense.

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

> *I shall certainly admit a system as empirical or 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*[6].

This means, that it is not necessary to be able to prove that the claim is true, but it must be possible to refute it by an empirical test. Finding claims about programming languages that do not fail this criteria is difficult – a claim about mathematical model cannot be refuted by experience; a claim about usability of language passes, but it can hardly resist refutation for a long time.

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 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*[7].

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[8]. Let us consider now some alternative treatments.

[4] First section in Gold and Simons (2008) 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).

[5] Perhaps because the predominant paradigm dictates that such claim is "unscientific" when based just on the author's introspection.

[6] Popper (2005), 18

[7] Ibid., 11

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

The new experimentalism

I will return to the problem of devising scientific theories about programming languages in the next section, but there is also another solution. Rather than defining what claim is scientific, we can sidestep the problem and 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[9] 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[10]. However, a group of philosophers[11] that A. F. Chalmers and others call *new experimentalists* believe that theory-independent experiments can form a foundation of scientific method:

> *According to its proponents, experiment can, in the words of Hacking (1983) 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*[12].

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 of experiments also means that radically different theories can be compared by comparing the (theory-independent) experimental knowledge that they enable.

The question that programming language researchers who want to subscribe to the "new experimentalist" approach need to ask is, how do we produce theory-independent experiments? In physics, an example of such experiment is Faraday's motor[13] – the experiment is relatively easy to replicate, it is not fallible (it usually works) and its effects are obvious to any scientist or even non-scientist, regardless of a theory they follow.

I will return to this topic later in the essay when discussing what we can learn from philosophy of science. To outline the claim that I will make later – I believe that, for programming language design (other than purely mathematical), medium-scale practical case studies can provide such theory-independent experiments. A sample application is an artifact that PL researchers as well as practitioners can understand and gain insights from.

Against method

I started this essay by trying to fit programming language research with the commonsense view about science. The new experimentalist approach discussed in the previous section gives a way to sidestep the issue. However, focusing on such narrow aspect of science (experiments) ignores vast amount of other valuable scientific practice.

Programming language research does not seem to easily fit with the commonsense view of science. While I will discuss more sophisticated structures shortly, let me avoid future disappointment. Even traditional sciences like physics 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[14].

In summary, Feyerabend says that scientific ideas are developed in much less organized manner than what its image suggests. Science often advances by proposing theories that contradict with established experimental results[15] and relies on ad-hoc approximations[16].

Feyerabend demonstrates his theory using a number of historical examples, including the Galilean revolution – the theories developed by Galileo are in direct conflict with scientifically accepted "facts" of his time. In addition to intellectual reasons, Galileo also uses propaganda[17] to change such established natural interpretations.

[9] Chalmers (1999), 36

[10] 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-independent experiment that yields new insights for a wider scientific community.

[11] Pioneering work in this direction is Hacking (1983)

[12] Chalmers (1999), 194

[13] Ibid., 196

[14] Feyerabend (2010), 12

[15] Ibid., 13

[16] Ibid., 43

[17] Ibid., 61

The purpose of the brief example is to illustrate 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.*[18]

Such view of science is very subjective and does not provide any guidelines for distinguishing "good science" and "bad science". This is an interesting point for programming language researchers – many languages used by the industry do not qualify as "good science" according to commonsense PL research perspective. Yet, they are popular and widely used. The anarchistic perspective might give us some hints on how to take such languages into consideration and perhaps learn from them, even though they are not originate from the scientific method.

However, this does not mean that we should study and read everything ever written. Feyerabend's comment on the selection procedure is 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*[19].

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.

Some philosophers of science, among them Chalmers, try to find a middle ground. They argue that there are some scientific standards, but these can change (which leaves enough room for the "anything goes" method). I will draw an inspiration from this approach later in the essay when I discuss how understanding the context of research work is important for its evaluation. Before that I briefly look at two influential works that attempt to capture the structure of scientific development by looking at the history of science.

[18] Ibid., 1
[19] Ibid., 163
[20] Chalmers (1999), 108

Structure of programming language research

The methodology followed by Kuhn and Lakatos is to look at the history of science and propose theories of philosophy of science that can capture common cases of scientific practice (such as Galilean and Newtonian revolutions in physics). Unlike Popper's falsificationism, such theories 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 philosophers of science. However, such treatment of science is also interesting because it suggests what hidden assumptions scientists commonly make – what aspect of science remains hidden during regular scientific practice. This section briefly summaries the work of Kuhn and Lakatos. I return to the hidden assumptions in the second part of this essay.

Scientific revolutions

According to Kuhn, science (after initial pre-scientific period) proceeds in cycles where period of *normal science* is followed by crisis and by 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*[20].

In Kuhn's perspective, the paradigm dominates and entire field (or a subfield). The existence of such paradigms 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*[21].

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"*[22].

The only moment when scientist become aware of the assumptions dictated by the paradigm is during the period of *crisis*. That is, when the paradigm is found insufficient for

[21] Kuhn (1970), 19
[22] Chalmers (1999), 112

solving problems (puzzles) within the normal science. In that case, the predominant paradigm is replaced with another:

> *[S]cientific revolutions are here taken to be (...) non-cumulative developmental episodes in which an older paradigm is replaced in whole or in part by an incompatible new one*[23].

When studying programming language research from the Kuhnian perspective, we try to identify different paradigms and their background assumptions. I do not attempt to do that in this essay. However, I would like to briefly consider the background assumptions commonplace in programming language 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 – as mentioned earlier, we believe that such models provide useful insights. I do not want to doubt the usefulness of such models, but the amount of trust in models is surprising when the aim is often to produce much larger industrial-scale implementations[24].

Aside from common assumptions, the paradigm also provides technique 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 predominant PL research paradigm has been easily ignored in the programming language Dart[25].

Research programmes

Another attempt to explain the structure of science has been made by Lakatos[26]. He looks how falsification (in the sense of Popper) is actually present in the history and notes that failure of a theory can be always ascribed to different aspects of the theory. This means, 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 the failure to auxiliary assumptions.

This is the basis for Lakatos's theory of research programmes. Similarly to paradigms, research programmes specify the background assumptions of a theory. 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 (...) might be. Lakatos referred to the fundamental principles as the* hard core *of a research programme*[27].

The hard core is its defining characteristic of a research programme. It is augmented with *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.

Philosophers of science studying the history of programming languages could now identify the different research programmes that PL researchers subscribe to. This is not the goal of my essay, but consider for example the development of pure functional programming. The hard core is formed by concepts such as immutable variables and data structures, 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 (e.g. an insufficient optimization in the compiler) and never to the hard core assumptions.

According to Feyerabend, Lakatos's methodology is so lax that it can accommodate almost everything[28]. Indeed, 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 only reconstructed in hindsight.

Nevertheless, the philosophy still provides some structure. It might not rule out "bad science"[30], but I argue that it provides a way to navigate through the complex web of programming language research that is otherwise ruled by the "anything goes" methodology. I return to this topic in later part of this essay and argue that understanding and acknowledging the research programme that a programming language researcher subscribes to can improve the practice of our field.

[23] Kuhn (1970), 92

[24] 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 realized by Sedlacek who writes: "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 (2011), 288)

[25] See Brandt (2011) for explanation and the motivation

[26] See Lakatos (1975) and Chapter 9 in Chalmers (1999)

[27] Chalmers (1999), 131

[28] Ibid., 154

[29] Ibid., 144

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

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. In this section, I briefly mention mathematics and also social sciences.

Answering the question whether mathematics is a science is equally difficult as answering the same question about programming language research. In his essay about future of mathematics, Devlin uses the following definition:

> *[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 will sound 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 factor that links 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].

This leads me to economics, which is the last subject that I want to draw inspiration from. In his recent book, Sedlacek rethinks economy from a much broader perspective that does not focus just on mathematical models, but includes long history of myths and religions:

> *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 form 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 aims to solve the problems that are faced by the IT community, we tend to dismiss "commonsense wisdom" of everyday software developers as unscientific. I believe we are often throwing out the baby with the bath water. I will suggest some ideas how to remedy this problem in the next section.

Learning from philosophy of science

The previous few pages were, following Feyerabend's criteria for selection, a very subjective exploration of some of the major works in philosophy of science. I made a number of digressions to suggest how the concepts relate to the research in programming languages, but I did not elaborate. This section goes in deeper detail. I present some (again, subjective) answers to the question: What can programming language research learn from the philosophy of science?

First, we've seen that 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. I believe that we need to accept this fact and make our practice more flexible to support the development of competing theories.

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

Third, programming language research develops numerous competing theories (programming languages) that are difficult to directly 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.

[31] Devlin (2008), 293.

[32] Turner, Eden (2011) discuss abstraction from the philosophical perspective

[33] Chalmers (1999), 47

[34] Sedlacek (2011), 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 (1999), 156).

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 some scientific standards. To quote Feyerabend:

> *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 (as we cannot know what the standards are until we look at science in hindsight), we can certainly identify a number of different standard in programming language research. Some work is focused on theory (with proofs) while other emphasises practical implementation (with performance measurements), but I believe the classification can be more fine-grained.

One language feature or PL theory often has different interpretations and, depending on the interpretation, researchers have different goals. This is essentially summarized by Lakatos's research programmes – depending on the research programme, a programming language researcher will work by modifying different auxiliary assumptions.

I argue that it is important to understand (and perhaps explicitly state) the research programmes that we subscribe to. This includes the *hard core* assumptions as well as methodology used to solve problems. This way 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, we judging a new work in the PL research from the perspective of an existing theory may rule out result that is important from a different perspective. 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 hypotheses that subscribes to an established theory from the perspective of *multiple different* established theories they may still be unacceptable if

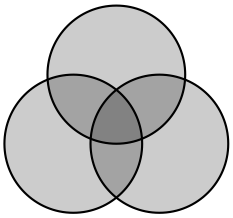

Figure 1. Overlapping cores of competing research programmes

they modify the hard core of one of the competing 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.

As discussed by Feyerabend, historical evidence shows that hypotheses inconsistent with established theories and 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 plurality of multiple competing theories.

If we openly allow multiple research programmes (and 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 summarized by Feyerabend – *"science is much more 'sloppy' than its methodological image"*[38].

New theories often take step back and do not necessarily have increased content (they do not add new results), but they define new problems or give a new perspective. We can easily find historical evidence from the programming language field where rejecting a novel theory early, because it suffers from problems that are 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 *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].

[36] Feyerabend (2010), 242
[37] Ibid., 20
[38] Ibid., 160
[39] Ibid., 112
[40] Ibid., 112

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

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, I argue that philosophy 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 about Kuhn and Lakatos, respectively, 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 shows that such early phases of scientific hypotheses or a research programmes are often long. I believe that the same is the case for programming language research and I argue that ignoring imperfect or unmathematical versions of research is not beneficial for our 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. As for early confirmations, I believe that successful case studies, showing the practical applicability of a language in some non-trivial domain, can play an equivalent role (more details are discussed in the next section).

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 early requirement of clarity:

> *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 the above statement and believe that programming language research needs to start with a 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].

Going back to programming language research – I mentioned case studies as one possible form of early confirmations of a theory and as a form of theory-independent experiments. The following section discusses the idea in detail.

[41] Feyerabend (2010), 8
[42] Chalmers (1999), 106
[43] Ibid., 135
[44] Ibid., 106
[45] Feyerabend (2010), 200
[46] Sedlacek (2011), 321
[47] Detlefsen (2008), 9 discussing Jaffe, Quinn (1993)

Standalone experiments: 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) 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. However, this is not a sufficient condition for a good experiment:

> *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 – but these claims need to be explicitly state, be it performance or the ability encode some mathematical pattern without cognitive overhead.

Moreover, 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].

As mentioned earlier, I argue that case studies provide a way to avoid the theory-dependence in programming language research. Let us examine the following definition of case study:

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

The most common artifact provided when discussing a novel programming language, language feature, tool or methodology is an implementation (e.g. a compiler, library or the tool). 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 real-world applications) and the analysis of such implementations. I believe that we need to accept that programming language research is not just mathematics and learn from social sciences – we should not seek precision where it is impossible and restrict our view to problems that have a precise mathematical formulation. Instead, 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:

> *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 programming language research. Artifact such as compiler implementation is clearly insufficient for comparison of multiple theories, but case studies that describe implementation of a non-trivial application, reflect on the experience and analyse different aspects of the development can be compared. A competing (incompatible) theory can attempt to solve the same problem and contrast the experience with earlier work – even if the comparison is going to be more subjective than in formal mathematical treatment.

[48] For example, see Feitelson (2006)
[49] Hauswirth (2013a)
[50] Hauswirth (2013b)
[51] Chalmers (1999), 37
[52] Ibid., 197
[53] Simons (2009), 21
[54] Chalmers (1999), 205

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].

I believe 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:

> *[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, I argue that we need to focus on theory-independent experiments and I propose case studies as such experiments. 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 programming language research field. In other words, we should not "*discard the immense treasures of knowledge and wisdom that are contained in the traditions*"[57].

Conclusions

In this essay, I tried to shed some light on the question: "*What can programming language research learn from the philosophy of science?*"

I started with a subjective exploration of some of the major 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.

References

Brandt, E. (2011). *Why Dart Types Are Optional and Unsound*. Online at: http://www.dartlang.org/articles/why-dart-types

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

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.

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.

Feyerabend, P. (2010). *Against method*. Verso (4th edition). ISBN 1844674428.

[55] Feyerabend (2010),27

[56] Meyerovich, Rabkin (2012)

[57] Lorenz (1984) as quoted by Feyerabend (2010), 131 footnote 16

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

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

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

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

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

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

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

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

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.

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

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

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*.

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

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

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

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

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

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/