Unpublished draft, April 2015

Against the definition of types

Tomas Petricek
University of Cambridge, United Kingdom
tomas@tomasp.net

Abstract

"What is the definition of type?" Having a clear and precise answer to this question would avoid many misunderstandings and prevent meaningless discussions that arise from them. But having such clear and precise answer to this question would also hurt science, "hamper the growth of knowledge"[1] and "deflect the course of investigation into narrow channels of things already understood"[2].

In this essay, I argue that not everything we work with needs to be precisely defined. I explore how the meaning of types is understood in programming language communities, how the meaning changes and why this is good for science and, finally, how can we think about this imprecise notion of 'types'.

> *Science is much more 'sloppy' and 'irrational' than its methodological image.*
>
> *(Against Method, Paul Feyerabend)*[3]

Introduction

Probably no other term in programming languages attracts as much attention and arguments as 'types'. A type is supposed to be a well-defined formal entity, but instead, it is a vaguely defined term with emotions attached to it.

Those with negative emotions towards types will blame types for failures that may not be caused by any fundamental property of types. For example, you could blame types for the verbosity of Java, but an ML user familiar with type inference will quickly object.

On the other hand, the proponents of types will often praise types for properties that are not essential for types and can be achieved in other ways. For example, editor support (e.g. auto-completion) can be attributed to types, but there are systems providing similar features not directly based on types.

We disagree even when we're on the same side of the barricade. For example, traditional arguments for types have been language safety and more efficient compilation. The recent TypeScript language adds 'types' to JavaScript, but its type system is intentionally unsound (hence no language safety) and types are erased when code is translated to JavaScript (hence no increased efficiency). Is it still a 'statically typed' when it is unsound? And has the purpose of types silently changed here?

In a recent essay about types Stephen Kell distinguishes between 'expression types' (following a logical tradition) and 'data types' (following an engineering tradition) and argues:

> *It would be nice to rewind history and choose some other word than "type" for some or all of the various meanings it enjoys today. Perhaps we can at least take greater care to qualify our usage (...)*[4].

I agree that being more explicit about what we mean when we have a clear idea in our mind is beneficial. However, the situation is much more complicated. The meaning of 'types' changes (and we usually do not notice), there are several communities that use the term differently and require different properties (often without a clear common ground).

If we look through the aura of perfection surrounding science, we'll find that this is not an uncommon situation. And in fact, many philosophers of science argue that it is healthy and necessary state of affairs.

In this essay, I argue that 'types' do not have to be defined. I discuss the issue from the perspective of philosophy of science, looking how similar issues have been treated in the contexts of mathematics, philosophy of language and natural sciences.

I start with a discussion of how the meaning of types differs between communities and how it changes. Then I look for arguments supporting the idea that types should be left undefined and, finally, I discuss some options for living in such unsatisfying (but realistic) world without exact definitions.

[1] Lakatos (1976), 74
[2] Feyerabend (2010), 200
[3] Feyerabend (2010), 160
[4] Kell (2014)

How the meaning of types changes

I start the essay by looking at various uses of the term 'type' in different eras and in different programming language communities and by exploring how incompatible the different uses are. This serves as a background for the next two sections that discuss philosophy of science theories that can explain such meaning incompatibilities and demonstrate that this is, in fact, quite common situation in science.

I would like to start with a personal anecdote. I was recently supervising the Types course and the lecture notes describe the uses of types and type systems as follows[5]:

- Detecting errors via type-checking, statically or dynamically
- Abstraction and support for structuring large systems
- Documentation
- Efficiency
- Whole-language safety

Fortunately, my students were already indoctrinated[6] and did not question the list. Indeed, past programming language used types for *all* of the reasons above. Nowadays, types are used for *some* of the reasons, but typically not all of them. As already mentioned, TypeScript sacrifices safety and efficiency and uses types mainly for documentation and (some) error detection. On the other hand, Julia[7] uses types exclusively for efficiency.

Arguably, types can even be used for none of the reasons above (consider proving mathematical theorems using Coq). And if we look further in history, types were invented by Russell to avoid paradoxes in foundations of mathematics. No doubt, they would be surprised by the list!

In the rest of this section, I'll explore a number of uses of types in more detail. I focus on uses of types arising from the 'logical tradition'.[8] We can find significantly different notions of 'type' even within such restricted area.

From foundations of mathematics to the lambda calculus

Types, in a sense similar to the one used in programming, were introduced by Russell in a paper Mathematical logic as based on the theory of types (Russell, 1908). However, Russell used types as a mechanism for avoiding paradoxes of the kind "class of all classes that do not contain themselves as elements."

The paper introduces a hierarchy of types such that propositions containing variables of type n are assigned type $n+1$. The theory of types avoids contradictions arising from self-reference as follows:

> *If we now revert to the contradictions, we see at once that some of them are solved by the theory of types. (...) Thus when a man says "I am lying", we must interpret him as meaning: "There is a proposition of order n which I affirm and which is false". This is a proposition of order $n+1$; hence the man is not affirming any proposition of order n; hence this statement is false and yet its falsehood does not imply (...) that he is making a true statement.* [9]

Here, the purpose of types is very different from how we use them in programming. Moving closer to programming, types were later introduced into λ-calculus.

Note that it would be incorrect to see the original λ-calculus as a simple programming language. It appeared (together with the theory of recursive functions and Turing's machines) as an attempt to formalize 'effective computability', that is a class of computations that can be carried out by mechanically (by a human) following a set of rules[10].

Church extended the λ-calculus with types in 1940 and outlines the purpose of his paper as follows:

> *The purpose of the present paper is to give a formulation of the simple theory of types which incorporates certain features of the calculus of λ-conversion. (...) the present partial incorporation has certain advantages from the point of view of type theory and is offered as being of interest on this basis (whatever may be thought of the finally satisfactory character of the theory of types as a foundation for logic and mathematics).*[11]

As we can see, Church views λ-calculus more as a work on the foundations of mathematics than as a method for describing computations. For later developments, it is worth noting that his system includes two base types; o for propositions and ι for individuals. Church does not define what these denote:

> *We purposely refrain from making more definite the nature of the types o and ι, the formal theory admitting of a variety of interpretations in this regard.*[12]

[5] Pitts (2015)

[6] *What's really demanded in the Church of Reason is not ability, but inability. Then you are considered teachable. A truly able person is always a threat.* Pirsig (1999), 392.

[7] Julia documentation (2015)

[8] The 'logical tradition' has been identified by Kell (2014)

[9] Heijenoort (1990), 166

[10] As discussed by Priestly (2011), 75, Turing's machines were more successful from this perspective, because the definition had greater 'psychological fidelity' (i.e. was closer to human intuition of how computation is done).

[11] Church (1940)

[12] Ibid

Although this notion of types is formally close to the notion of types in programming languages, the intuition behind types is different – Church did not see types as "sets of possible values", which is a more recent view discussed next. In summary, in the early λ-calculus which was to become a model of programming languages later, types first appeared in a sense that is very different from their modern use.

From expression types to computation types

I'm not attempting to write a complete historical account of the development of types, so let's jump forward to types in early ML languages. At this point, it was common to see types as sets of possible values that a computation can produce. For example, Cardelli describes types as follows:

> *There is a universe V of all values containing simple values like integers, data structures (...), and functions. This is a complete partial order (...), but in first approximation we can think of it as just a large set of all possible computable values. A type is a set of elements of V.*[13]

This later development is interesting as it combines the 'logical tradition' (from λ-calculus) with the 'engineering tradition'[14] (using integers, reals, records and other data types). We could view Church's simply typed λ-calculus through the new perspective of "types as sets", but that would be misinterpreting the original work. In other words, there is a small and subtle change in how we think about types. And we might not even notice the change if we are not explicitly searching for it!

The subtle change in meaning of types affects not just how types are understood, but also how they are used and what are the important properties required from them. In "Types and Programming Languages", Pierce adopts a similar set-based view and defines what a type system is as follows:

> *A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.*

This book is the standard text on types nowadays, so this is the view we teach today. It follows the syntactic approach to type soundness introduced by Wright and Felleisen[15]. The approach works extremely well for systems that understand types as sets of values produced by a computation.

However, this is not the end of the story. Another slight shift in the meaning of types comes with the development of type and effect systems and monads[16]. Here, the type captures not just the set of produced values, but also information about other effects that the computation has.

Consider the following example, which uses two reference cells r and s allocated in separate memory regions ρ and σ, respectively and assigns the value from s to r:

$$r\colon \mathrm{ref}_{\rho}, s\colon \mathrm{ref}_{\sigma} \vdash r := !s : \mathrm{unit} \,\&\, \{\mathrm{write}\ \rho, \mathrm{read}\ \sigma\}$$

Here, the type and effect of the expression tells us that the computation returns a value of type unit (which is quite uninteresting singleton set) and also writes to a memory region ρ and reads from a memory region σ.

When we consider effect systems, thinking of types as sets of values is leaving out an important part of the story. It is a subtle change, which makes us think that some small extension should be enough to fix the mismatch. However, we can also see this as a completely different meaning of types – perhaps not in all contexts and all communities, but certainly in some.

For example, in his talk in 2014, Benton argued that types should be interpreted as relations:

> *Express meaning of high-level types as relational, extensional constraints on the behaviour of compiled code*[17]

In this view, the type of the above expression specifies that, for all memory regions, the value after performing the computation is the same as the value before, with the exception of the region ρ. This view also changes the purpose of types. To quote Benton *"Types are about abstractions not about errors"*.

In this section, I looked at the meaning of 'types' in a quite narrow area of current research that follows from the logical tradition. However, even if we use such restricted perspective, the meaning and the purpose of types can be very different. In fact, the Types course from which my earlier list was taken, was taught in the same year and room as Benton's talk.

Unsound, relatively sounds and super sound type systems

So far, the story of types has mostly been linear, so we could think that there is an ultimate perfect notion of types that we are slowly getting closer to. However, this is also not the case. Three developments in the recent years complicate the matters and evolve types in very different directions.

The first development that I discuss in this (incomplete) survey is dependently typed programming. Dependent typing can be viewed from a logical tradition as an extension of the correspondence between types and mathematical logic:

[13] Cardelli, Wegner (1985)
[14] The terms 'logical' and 'engineering tradition' are due to Kell (2014)
[15] Wright, Felleisen (1994)
[16] Lucassen, Gifford (1988)
[17] Benton (2014)

> *Generalizing the [Curry-Howard] correspondence to first-order predicate logic naturally leads to dependent types*[18]

Here, the notion of types goes back to Russell and Church and types appear as constructions of mathematical logic, interesting for their formal properties. Following this direction, types have been used in logical frameworks and theorem proving.

Dependent types can be also viewed from the engineering tradition, as another way to make types more precise and rule out more of the possible errors:

> *For instance, the type of an array might include (...) the size of the array, making it possible to verify absence of out-of-bounds accesses statically. Dependent types can go even further than this, effectively capturing any correctness property in a type.*[19]

This use of dependent types is emphasizing the error detection and whole-language safety properties. The array example could be viewed through the "types as sets" perspective, but this fails for the "any correctness property" generalization.

Dependent types unify the original logical tradition with the modern programming understanding of types. Arguably, the unification is imperfect as the two traditions use different terminology, methods and also use types for different purposes (proving vs. programming).

Second development is taking types in very different direction. TypeScript and Dart are two languages that both compile to JavaScript and both have an unsound type system. They are not *unsafe*, because the execution engine checks types dynamically. Here, the focus of types is shifting from provable correctness to documentation and tool support. According to Bracha, types in Dart provide the following benefits:

- *Documentation for humans. It is much easier for people to read your code if it has judiciously placed type annotations.*
- *Documentation for machines. Tools can leverage type annotations in various ways (...).*
- *Early error detection. Dart provides a static checker that can warn you about potential problems (...)*[20]

First two of the points view types as documentation, either for humans or for machines or to enable tooling such as navigation and auto-completion. Types in Dart are not unlike types in other modern programming languages, but we can see another shift in their meaning and types.

In Dart and TypeScript, types do not "mean anything" in the sense discussed by Cardelli or Benton. They are still used for (limited) error detection, but the main focus is shifting to documentation and tool support.

Finally, the third development comes with type providers in F# and Idris[21]. Type providers extend the type system with the ability to programmatically generate types based on external data. For example, the World Bank type provider imports country names and indicators as statically checked members (fields) of a type. This is sound as long as the external data source does not change. To reflect this, languages with type providers have *relativized* type safety:

> *The relativized safety property does not guarantee the same amount of safety as standard type safety for programming languages without type providers. However, it reflects the reality of programming with external data sources (...).*[22]

Type providers are interesting because they arise from the ML tradition of languages. Unlike TypeScript or Dart, F# is very strict about its type system in certain ways. Yet, it develops the notion of type in a different way, which can be viewed as unsound from a certain classical perspective.

In context of type providers, types serve both for error checking (although with a relativized twist) and as a documentation for human and the machine (to provide auto-complete). Following the ML tradition, types can here be viewed as "sets of values" with a syntactic safety property, but their purpose is closer to that in Dart and TypeScript.

Summary

This incomplete review shows that types are not a single well defined concept. If we look at the entire history of 'types', both the meaning and the purpose of types changed dramatically. This often happens in small steps where a new purpose or a meaning is added to existing ones, but later becomes dominant.

Church refrained from defining the meaning of base types; later, types were viewed as "sets of values", but this interpretations does not hold when types begin to include effects of a computation. Along the way, the purpose varied between logical, error-detection and documentation.

A believer in a certain tradition (myself included) can state that their meaning of types is the right one and others are just misuses. But why should a certain tradition be the right one in such a diverse and evolving space?

[18] Aspinall, D., Hofmann (2005), 48

[19] Chlipala (2014), 8

[20] Bracha (2011). The article also mentions possible performance improvements. To my understanding this has not been the focus of Dart recently.

[21] Syme et al. (2013), and Christiansen (2013)

[22] Petricek et al. (2015)

Is inconsistent and evolving meaning harmful?

From a rational scientific perspective, my presentation of types might be disappointing. How can science progress, if we cannot agree on a consistent meaning of basic terms we work with? And how can our work improve, if the purpose keeps changing without us even noticing?

In this section, I discuss two theories that explain situations very similar to those that we can see with types. First, *research programmes* give a view of science where multiple inconsistent approaches coexists. Secondly, *concept stretching* from philosophy of mathematics explains how our intuitive understanding of entities can evolve.

Inconsistent theories and research programmes

The *consistency condition* states that developments of scientific theories should be consistent with previous work. As discussed in the previous section, this is not always the case with the work on types. According to Feyerabend, this is a common situation and we must therefore look at science differently:

> *[T]he methodological unit to which we must refer [is] a whole set of partly overlapping, factually adequate, but mutually inconsistent theories.*[23]

The idea that science consists of multiple sub-structures was first advocated by Kuhn with his *research paradigms*. Research paradigms capture the shared assumptions and methods of scientists operating in a certain field. A paradigm changes through *paradigm shifts* when a new, inconsistent set of assumptions and methods replace the old ones. However, research paradigm applies to the whole community and so it does not explain the local inconsistencies with types.

A less widely known theory that is closer to Feyerabend's above view and is also more applicable to types is Lakatos's theory of *research programmes*. In this view, science is formed by multiple competing research programmes. Each research programme is formed by a *hard core* consisting of assumptions that are never doubted and auxiliary *protective belt* that can be freely modified:

> *[Some laws or principles] are not to be blamed for any apparent failure. Rather, the blame is to be placed on the less fundamental components. A science can then be seen as the pragmatic development of the implications of the fundamental principles.*[24]

This theory stats that science proceeds in a rational way, but only within a research programme. If we judge the work done in one research programme through the perspective of another one, we can find it inacceptable – work in another research programme will often break fundamental assumptions that we subscribe to and will use methods that we do not accept.

I believe we can use the perspective of research programmes to shed some light on types in programming language research. Looking at the examples discussed in the previous section, we can identify at least two different programmes:

- The definition of type system by Pierce captures the core assumptions of one research programme. According to it, a type system must be sound, tractable and serves to detect errors. The programme also provides standard tools such as syntactic approach to type soundness.
- According to the programme advocated by Benton, types should have a meaning. The methods of the programme include denotational approach to semantics.
- According to another research programme (including Dart, TypeScript and, to some, extent F#), types should improve the usability of a programming language. Its proponents are willing to sacrifice other properties, including whole-language safety which is crucial in other communities. The methods include e.g. using types in editor tooling.

Describing the research programmes precisely in detail is work that I leave to philosophers of science. My main point is that looking at our field through this perspective is useful and can help us understand how concepts such as types are used and why we often fail to find a shared understanding. It is simply because we subscribe to different core principles.

Should we then identify the different research programmes and (as Stephen Kell suggested) use different, unambiguous naming in each of them? I believe there is more to types. In particular, they are what sociologists call boundary objects:

> *Boundary objects are objects which are both plastic enough to adapt to local needs and constraints of the several parties employing them, yet robust enough to maintain a common identity across sites.*[25]

This definition fits well with how types are used in programming. They are used differently by different communities (following different research programmes), but we are not talking about completely different things! Hence, it makes sense to use

[23] Feyerabend (2010), 20
[24] Chalmers (1999)
[25] Star, Griesemer (1989)

a common name for types across multiple research programmes. As boundary objects, types are actually valuable entities:

> *They have different meanings in different social worlds but their structure is common enough (...) to make them recognizable, a means of translation. The creation and management of boundary objects is key in developing and maintaining coherence across intersecting social worlds.*[26]

In other words, types let us translate interesting ideas between different research programmes. For example, tooling that was developed based on types in Java-like languages (auto-completion) has been adapted and used for writing proofs in dependently typed languages, despite having very different notion of type under the cover.

How meaning changes through concept stretching

In his book Proofs and refutations, Imre Lakatos tells the story of Euler characteristic of polyhedra ($V - E + F = 2$, where V, E, F are the numbers of vertices, edges and faces). Lakatos describes how mathematicians face numerous counter-examples that were discovered (such as nonconvex polyhedra, polyhedra with tunnels etc.).

Lakatos introduces the notion of *concept stretching*, which happens when a new counter example (of a previously inconceivable form) is discovered:

> *Then came the refutationists. In their critical zeal they stretched the concept of polyhedron, to cover objects that were alien to the intended interpretation.*[27]

Concept stretching takes a concept and extends it to include a new idea that is not explicitly ruled out by the formal definition, but is of a novel form and has not been considered before.

Concept stretching also happens in the context of types. One example is using types to capture effects of a computation. This extends the idea of a type, but it also accidentally breaks standard interpretations of types (types as sets of values) and complicates the standard methods (syntactic soundness).

In Lakatos's story, there are monster-barrers who try to save the original interpretations and methods by labelling the newly discovered counter-examples as monsters that should be ruled out. However, this does not work:

> *The curious thing is that concept stretching goes on surreptitiously: nobody is aware of it, and since everybody's 'coordinate-system' expands with the widening concept, they fall prey to the heuristic delusion that monster-barring narrows concepts, while in fact, it keeps them invariant.*[28]

The fact that concept-stretching happens secretly is interesting for our discussion about types too. For example, I believe that the shift from Church's original simply typed lambda calculus to types in simple functional languages is certainly larger than is generally understood.

The introduction of unsound type systems is another example of concept stretching. Just like adding tunnel through a polyhedra, it extends the concept of type system in a previously inconceivable direction. In this case, a large part of the programming language community reacts as *monster-barrers* from Lakatos's story. That is by labelling unsound type systems as monsters and refusing to admit them into a well-behaved society. It is not difficult to find modern variations on a quote that appears in Charles Hermite's letter from 1893:

> *I turn aside with a shudder of horror from this lamentable plague of functions which have no derivatives.*

Against the definition of types

In the first part of this essay, I gave a few examples of how the notion of 'type' was understood by different communities in different times. Then, I suggested how we could find structure in this development using two concepts from philosophy of science: *research programmes* and *concept stretching*.

The reader might expect that I'll now say that we should take extra care when talking about types, document our research programme and watch carefully to avoid (or acknowledge) concept stretching. In other words, we should analyse, properly name and classify our terms.

Doing all this is, indeed, a useful contribution to science, yet I strongly argue that we should not *require* more order when discussing types. We should be flexible enough to accommodate people such as Phaedrus from Pirsig's Zen and the Art of Motorcycle Maintenance who identifies Aristotle as the founder of the modern scientific approach and laments:

> *Phaedrus saw Aristotle as tremendously satisfied with this neat little stunt of naming and classifying everything. (...) he saw him as a prototype for many millions of self-satisfied and truly ignorant teachers throughout the history who have smugly and callously killed the creative*

[26] Ibid

[27] Lakatos (1979), 84

[28] Lakatos (1979), 86

spirit of their students with this dumb ritual of analysis, this blind, rote, eternal naming of things.[29]

Pirsig's wording might be a hyperbole, but there is some truth in it. Creative uses of types and other concepts often break some of the established rules and principles of the time and we only find a way to reconcile them in retrospect.

Paul Feyerabend discussed this idea from the perspective of philosophy of science with historical grounding. In the next two sections, I look at the points he makes and how his observations follow from history of science.

Epistemological anarchism and clarity of terms

As discussed earlier, I believe that we can search for clarity, especially in retrospect, but we should not *require* it. The problem is that clarity means a different thing in retrospect and when new ideas are created. Paul Feyerabend explains how the requirement of clarity restricts and changes our thinking:

> *[T]o 'clarify' the terms of a discussion does not mean to study the additional and as yet unknown properties of the domain in question which one needs to make them fully understood, it means to fill them with existing notions from the entirely different domain of logic and common sense, (...) and to take care that the process of filling obeys the accepted laws of logic.*[30]

New notions of type may not perfectly fit with the established understanding. Initially, this may not appear as a conceptual shift, but perhaps as a technical fault (that could be corrected). But this should not be a reason for rejecting them – we can accommodate the new notions, but only later once the *accepted laws of logic* evolve.

For example, when types were first used for the tracking of effects the work was not rejected, despite the fact that it did not clearly describe the structure of "set of values" that a type with effect annotation denotes. Perhaps one could invent an inelegant (but formally sufficient) answer, but this would shift the focus of the work in a different and much less interesting direction. Feyerabend continues as follows:

> *So the course of an investigation is deflected into the narrow channels of things already understood and the possibility of fundamental conceptual discovery is significantly reduced.*[31]

This Feyerabend's point beautifully describes why we should not strictly require clarity. Interesting developments (when new research programmes are born) often change the meaning, require the development of new methods and ways of thinking. Yet, these ideas can only be expressed using imperfect terms that are currently available.

We could argue for the claims made in this section based on humanitarian grounds (and Feyerabend did that too), but the more important point here is historical. If we look at the past developments in science, we can see that Feyerabend's *[epistemological anarchism] is more likely to encourage progress than its law-and-order alternatives*[32].

How science actually works

Feyerabend's position may be extreme for some readers, so I'll start by arguing for inexactness and imprecision in early stages of research programmes. Both Lakatos (speaking of research programmes) and also Kuhn (speaking of research paradigms) argue that early developments start with vague concepts and even ignore experimental failures:

> *Early work in a research program is portrayed as taking place without heed or in spite of apparent falsifications by observation*[33]

> *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*[34]

In early development of a research programme, the focus is on achieving something new (capturing effects of computations, providing better developer tools in dynamic environment), but other issues that are important for established science (what Chalmers calls *apparent falsifications*) can be ignored. In Kuhn's research paradigms, the situation is similar – paradigms emerge when current approaches start failing, but they emerge in imperfect forms.

However, the difficulty is noticing when a new research programme starts to emerge. This is easier to see in retrospect than during the development itself. Feyerabend summarizes this position with his famous slogan 'anything goes':

> *To those who look at the rich material provided by history and who are not intent on impoverishing it in order to please their lower instincts, their craving for intellectual security in the form of clarity, precision, 'objectivity', 'truth', it will become clear that there is only one principle that can*

[29] Pirsig (1999), 360
30 Feyerabend (2010), 200
31 Ibid, 200
32 Feyerabend (2010)
33 Chalmers (1999), 135
34 Chalmers (1999), 106

> *be defended under all circumstances and in all stages of human development. It is the principle: anything goes.*[35]

As Feyerabend later said, 'anything goes' is not a principle, but *the terrified exclamation of a rationalist who takes a closer look at history*[36]. And I believe that the complex developments of the notion of types outlined in the introduction also support this position.

Now, this does not mean that we should abandon all principles in all situations. This is not what Feyerabend advocates. When working within a well-developed area, it makes sense follow its principles and exact definitions that it provides:

> *We see that the principles of critical rationalism (...), though practiced in special areas, give an inadequate account of the past development of science as a whole and are liable to hinder it in the future.*[37]

The concepts of research programme and concept stretching that I discussed earlier in this essay provide a useful guidance. We should try to understand what research programme we are contributing to and follow its rules. However, we should also be aware of the fact that concepts can be stretched and that there are other programmes that follow different principles.

Summary

In this section, I presented philosophical arguments to support the main idea of this essay. That is, we should not require a precise definition of the notion of 'types'. Requiring clarity means that we can only talk about things we already understand – perhaps in greater detail, with more generality and in more elegant ways, but not about fundamentally new ideas.

There may be some structure locally. I believe that research programmes and concept stretching are useful for understanding some of the uses of 'types'. However, this can never capture the full complexity of scientific reality.

To end the essay on a more positive note, the following section discusses three ways of thinking about types that can be useful in the complex reality without exact definitions.

Living with undefined types

A type is not a formal concept that can have a precise definition. This can be the case in some narrow areas and we can use the precise definition *within* the narrow area, but how can we work with types if we want to operate and think outside of a particular research programme?

Philosophy of science describes a number of methods or ways of thinking that do not require precise definitions. I believe that these provide a useful complement to the rigorous methods that we use when operating within a narrow rigorous areas of an established research programme.

The following three sections look at three ways of thinking about types. These are based on how we *use types*, what are *conventional ideas* associated with types and what we can *do with types*.

Language games and how we use types

One way of understanding the meaning of a term without a precise definition is to look at the context in which it is used. Feyerabend suggested that this is how terms attain their meaning in early stages of theory development:

> *The terms of the new language become clear only when the process is fairly advanced, so that each single word is the center of numerous lines connecting it with other words, sentences, bits of reasoning, gestures which sound absurd at first but which become perfectly reasonable once the connections are made.*[38]

The philosopher who first claimed that "meaning is use" is Ludwig Wittgenstein. I believe that his ideas on language can suggest ways of dealing with undefined terms in science too. He describes the idea in Philosophical Investigations as follows:

> *For a large class of cases of the employment of the word "meaning" – though not for all – this word can be explained in this way: the meaning of a word is its use in the language.*[39]

Similarly, the meaning of a scientific term can be explained by its use in the scientific community. Types are used in many different contexts (proving program correctness, explaining the meaning of programs, providing tools for writing programs) and we can study how different forms of types can be used in different contexts.

Wittgenstein calls these contexts *language games* and notes that they are diverse and changing:

> *But how many kinds of sentence are there? There are countless kinds (...). And this diversity is not something fixed, given once for all; but new types of language, new language games, as we may say, come into existence and*

[35] Feyerabend (2010), 12
[36] Ibid, vii
[37] Ibid, 160
[38] Feyerabend (2010), 200
[39] Wittgenstein (2009), no.43

others become obsolete and get forgotten. (We can get a rough picture of this from the changes in mathematics.)

It is worth noting that Wittgenstein also relates his work to the changes in mathematics, so it seems reasonable to use his ideas for thinking about notions like types (which developed at the same time as Wittgenstein's philosophy).

What are the "language games" surrounding types and how they change? I believe that this is one interesting question we should explore to understand types! There is a lot of them: proving program properties with types, documenting developer intentions with types, improving performance with types and so on. The language games also change in time. For example, the "using types to build foundations of mathematics" language game has been at the birth of types at the beginning of 20th century, but has only regained prominence with the later developments of Per Martin-Löf's type theory.

However, explaining the existing language games is only one part of our investigation:

It is not the business of philosophy to resolve a contradiction (...), but to render surveyable the state of mathematics that troubles us (...). [W]e lay down rules, a technique, for playing a game and that then, when we follow the rules, things don't turn out as we had assumed.

To paraphrase the above quote, we do not need to resolve all the inconsistencies between different understandings of types. Instead, we can focus on creating interesting contexts in which the concept of a type can be used and explored.

What would be such language games for exploring properties of types? One example I can think of is the well-known puzzle referred to as *the expression problem*[40]. The expression problem reveals the abstraction and error-checking capabilities of a system and it has been used for looking at a large number of very diverse notions of type.

The expression problem gives a very specific perspective (just like some of Wittgenstein's language games), but it shows how we can talk about types without requiring a clear definition. To my best knowledge, there are not many puzzles or language games similar to the expression problem. I believe that we need to construct more language games to explore other properties of types for all the different areas in which types are used in programming.

Stereotypes and the meaning of types

Seeing programming language research through the perspective of competing research programmes explains why different communities view types differently, but it makes it difficult to say what the *meaning* of type is outside of the individual research programmes. Intuitively, we still have some overall idea about types, so saying that there is *no meaning* seems wrong.

One philosopher who addresses this question in the context of meaning of words is Hilary Putnam. However, the following motivation from Ian Hacking's book is a perfect fit for the problem addressed in this essay too:

[W]e need an alternative account of meaning which allows that people holding competing or successive theories may still be talking about the same thing.[41]

Putnam's theory is interesting because it gives us a way to talk about meaning in the real setting where different people talk about types, but using different perspectives. I find it useful as another example showing that we can think about things without precise definitions.

Hacking introduces Putnam's theory using an analogy with a dictionary. What would a dictionary definition for a programming language concept of type consist of?

A dictionary begins an entry with some pronounciation and grammar, proceeds past etymology to a lot of information, and may conclude with examples of usage.

Putnam's meaning is specified by four components – syntactic marker (type is a countable noun), semantic marker (a category to which type belongs, i.e. computer science entity), stereotype and extension (set of all things that are type).

The interesting part of the definition (and the part that is interesting for this essay) is stereotype:

[A] standardized description of features of the kind that are typical, or 'normal', or at any rate stereotypical. The central features of the stereotype generally are criteria – features which in normal situations constitute ways of recognizing if a thing belongs to the kind (...).[42]

This is a down-to-earth notion of meaning, but I believe that this is how many practitioners of the field think about types. We know what features are generally associated with 'types' and we can, certainly, use those to recognize a type.

Putnam illustrates the idea using tigers as an example. One such stereotype about tigers is that they are striped. But a white albino tiger is still a tiger. Similarly, a type is a classification of values that computations can produce. But a type that represents behaviour of computation is also a type. Similarly, a type is a decidable syntactic program property, but a type that

[40] The problem is extending a set of objects and functions in two directions – by adding new kinds of objects and new functions. Wadler (1998)

[41] Hacking (1983), 75

[42] Putnam, p230

cannot be effectively decided is still a type (and the list of stereotypical properties of types continues).

Another useful point made by Hacking is that illustrations in children books illustrate the stripiness of tigers to build the stereotype. Similarly, computer science textbooks discuss soundness of type systems to build a stereotype about types. But this does not necessarily mean that they give a full account of what a type is. Indeed, stereotypes are not exact definitions:

> *The fact that a feature (e.g. stripes) is included in the stereotype associated with a word X does not mean that it is an analytical truth that all Xs have that feature, nor that most Xs have that feature. (...) If tigers lost their stripes they would not thereby cease to be tigers.*[43]

Just like tigers can lose their stripes, types can lose some of their stereotypes. The stereotypes associated with the early notion of types included their use to avoid paradoxes, but also many other things (such as categorization of terms in a formula). The avoidance of paradoxes stereotype has been lost when types started to be used in programming languages, but other stereotypes associated with them remained. Similarly, properties that we ascribe to types now may not be representative stereotypes of types in the future.

When discussing Wittgenstein's language games in the previous section, I concluded with the suggestion that we should construct new language games to explore properties of types. Unlike language games, Putnam's theory does not suggest any new method of inquiry. However, I think that it is useful for another reason – it is perhaps the closest explanation to how computer scientists think about types. As such it makes explicit some of the aspects of meanings of types.

For example, it is useful to keep it in mind when reading about types in textbooks. Such textbook descriptions are really two things – formal definitions within the context of a narrow research programme *and* stereotypes for types in the broader sense. We should not be confusing the two!

Scientific entities and doing things with types

So far, this essay was focused more on how we think about types, but we can also take a practical attitude and look at *doing* things with types. The idea underlying this section is that we can do interesting things with types without having a full and developed theory of what types are.

In the context of programming languages, similar point has been made by Richard P. Gabriel in his recent Onward! essay:

> *[I]n the pursuit of knowledge, at least in software and programming languages, engineering typically precedes science (...) even if science ultimately produces the most reliable facts, the process often begins with engineering.*[44]

I agree with Gabriel that many interesting ideas in programming languages start with engineering or *experimentation*. This might be because experimentation in computing is very cheap compared to natural sciences – but, as a matter of fact, the same has been said about science in general.

Ian Hacking defends a very similar position, which has been labelled *new experimentalism*:

> *[I] make no claim that experimental work could exist independently of theory. (...) It remains the case, however, that much truly fundamental research precedes any relevant theory whatsoever.*[45]

I will not discuss the details in this essay. Hacking's excellent book provides a number of examples showing that *there have been important observations in the history of science, which have included no theoretical assumptions at all*[46].

Another interesting point made by Hacking is that it is the theoreticians who appear in the history books. This explains why we can easily recall authors of famous theories, but hardly remember any famous experiment and experimenters:

> *Before thinking about the philosophy of experiments we should record a certain class or caste difference between the theorizer and the experimenter. It has little to do with philosophy. We find prejudices in favour of theory, as far back as there is institutionalized science.*[47]

Despite the prejudices against the experimentalist approach to computer science (even the word *engineering* seems to have negative connotations in some circles!), I believe that it is an extremely valuable approach. And indeed, there are many systems that involve types which were not preceded by a full-scale theory, but provided useful and novel insights.

Type providers can be used as an example. They first appeared in the F# 3.0 language in 2011[48], but without a full theory that would be usual in theory-founded work. Yet, type providers already influenced other languages[49] and the theory explaining them started appearing too.[50]

An important question about experimentalist work in programming languages is, how do we observe the results of our experiments? (Here, I intentionally avoided using the term

[43] Putnam (1979), 250
[44] Gabriel (2012)
[45] Hacking (1983), 158
[46] Ibid, 175
[47] Ibid, 150
[48] Syme (2011)
[49] Christiansen (2013)
[50] Petricek (2015)

'evaluate', which suggests quantitative measurements; for experiments, it is sufficient to observe interesting results.)

According to new experimentalists, experimenting is not stating or observing, but *doing*. What matters is how scientific entities can be manipulated to cause other interesting effects Hacking uses electrons as an example, but we can similarly think about types:

> *[F]rom the very beginning people were less testing the existence of electrons than interacting with them. (...) The more we come to understand some of the causal powers of electrons, the more we can build devices that achieve well-understood effects in other parts of nature.*[51]

What can we *cause with types*? I think the new experimentalist perspective suggests an important point about programming language experiments. We can implement a compiler or a type checker for a given type system, but this is merely a different way of presenting the same theory.

When experimenting in programming languages, we need to create experiments that somehow interact with the outside world. Of course, we can measure some characteristic (performance, code complexity), but we can also observe how our experimental system can be used in practice.

I previously argued that one way of presenting such computing experiments is in the form of case studies[52], but I believe that this is an underexplored area with interesting possibilities. For example, the Future of Programming workshop[53] made it possible to submit (what I would call *experiment reports*) in the form of webcasts.

There are two more points about new experimentalism that I find relevant to work on programming languages and types. To quote Chalmers's introduction of the philosophy:

> *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. (...) [I]f scientific progress is seen as the steady build-up of the stock of experimental knowledge, then the idea of cumulative progress in science can be reinstated (...).*[54]

In science, isolating a stable and repeatable experiment is hard and experimentalists have practical ways for making reproducible experiments. Quite similarly, programming language experimenters or engineers have ways of producing systems that work in practice (now you can again see the prejudices against experimentalism; even the phrase *works in practice* is frowned upon). This is an important point – for example, some of the practical limitations of type providers limit their scope to an area where the mechanism works well[55]. But in theory-oriented work, such limitations would remove much of the complexities and subtleties that theoreticians find interesting.

The second important point that Chalmers makes is that new experimentalism makes it possible to recover the idea of cumulative growth of knowledge. As can be seen from my introduction, the notion of type is changing and so we cannot claim we are getting closer to a 'perfect' type. However, if we accumulate the experiments – practical problems that can be solved with types – we have a way of talking about growth of scientific knowledge.

To conclude this section, another way of working with types is to experiment and see what we can do with types. The history of science shows that this experimentalist approach is fruitful method. I also believe that we have a unique chance to find new and better ways of presenting experimental observations through formats such as webcasts.

Despite the prejudices against experimentalism in both science and computing, doing experiments is an important part of science and experiment have a life of its own[56]:

> *One can conduct experiment simply out of curiosity to see what will happen. (...) The physicist George Darwin used to say that every once in a while one should do a completely crazy experiment (...).*[57]

Conclusions

This essay was inspired by two things. First, by the frequent misunderstandings when discussing types and, second, by the calls for an exact definition of types to avoid such misunderstandings. My answer is that it is impossible to give a formal and generally acceptable definition of what a type is and that such definition is at odds with scientific reality. Rather than seeking the elusive definition that does not exist, we should instead look for innovative ways to think about and work with types that do not require an exact formal definition.

[51] Hacking (1983), 262

[52] Petricek (2014)

[53] Available at: http://www.future-programming.org/

[54] Chalmers (1999), 194

[55] For example, F# type providers can be parameterized by values of primitive types (integers, strings, etc.), but not by arbitrary types and, in particular, not by other user-defined types. This would extend the focus of the feature from data-access to meta-programming – this is equally interesting problem, but very different and more theoretically complex.

[56] Hacking (1983), xiii

[57] Hacking (1983), 154

To motivate the essay, I started with a brief and incomplete history of types. As the examples demonstrate, the meaning and the purpose of types is continuously changing and different communities have different core beliefs of what a type is. In philosophy of science, this is known as *concept stretching* and *research programmes*. If we look at the history, we find some structure and precise definitions *locally*, but this can never capture the full complexity of scientific reality.

It we want to talk about types outside of a narrow research programme, we need to find ways of dealing with types without a precise definition. I proposed three alternatives – those are based on how we *use types* (inspired by Wittgenstein's language games), what is the *conventional idea of type* (based on Putnam's stereotypes) and what we can *do with types* (inspired by Hacking's new experimentalism). I believe that these provide worthwhile methods of inquiry that can provide interesting insights into what *types* are outside of a narrow boundary delimited by a particular formal definition.

References

Aspinall, D., Hofmann, M. (2005). *Dependent Types*. In Pierce, B. C. (ed.) *Advanced topics in types and programming languages*. MIT press, 2005.

Benton, N. (2014). *What We Talk About When We Talk About Types (talk)*. Talk slides retrieved from: http://research.microsoft.com/en-us/um/people/nick

Bracha, G. (2011). *Optional Types in Dart*. Available online at: https://www.dartlang.org/articles/optional-types/

Cardelli, L., Wegner, P. (1985). *On understanding types, data abstraction, and polymorphism*. ACM Comp. Surv. vol. 17, n. 4

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

Christiansen, D. R. (2013) *Dependent type providers.* Proceedings WGP Workshop.

Church, A. (1940). *A Formulation of the Simple Theory of Types*. The Journal of Symbolic Logic, vol. 5, no. 2, pp. 56-68

Chlipala, A. (2013). *Certified Programming with Dependent Types*. MIT Press, ISBN: 9780262026659

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

Gabriel, R. P. (2012). *The Structure of a Programming Language Revolution*. In Proceedings of Onward! 2012.

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

Heijenoort, van, J. (1990). *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931 (Source Books in the History of the Sciences)*. Harvard University Press; New Ed edition. ISBN: 978-0674324497.

Pirsig, R. M. (1999). *Zen and the Art of Motorcycle Maintenance*. HarperCollins Publishers, ISBN 978-0-06-167373-3.

Pitts, A. M. (retrieved, 2015). Lecture notes on types. University of Cambridge. Available at: http://www.cl.cam.ac.uk/teaching/1314/Types/

Priestley, M. (2011). *A Science of Operations: Machines, Logic and the Invention of Programming*. Springer. ISBN: 978-1848825543.

Putnam, H. (1979) .*Philosophical Papers, Vol. 2: Mind, Language and Reality*. Cambridge University Press. ISBN: 978-0521295512

Star, S., Griesemer, J. (1989). *Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology*, Social Studies of Science, vol. 19, no. 3, pp.387–420

Julia documentation (retrieved, 2015). *Types*. Available at: http://julia.readthedocs.org/en/latest/manual/types/

Kell, S. (2014). *In Search of Types*. In Proceedings of Onward! Essays 2014.

Lakatos, I. (1976). *Proofs and Refutations*. Cambridge University Press. ISBN: 0-521-29038-4.

Lucassen, J. M., Gifford, D. K. (1988). *Polymorphic effect systems*. In Proceedings of POPL.

Petricek, T. (2014). *What can Programming Language Research Learn from the Philosophy of Science?* In Proceedings of the 50th Anniversary Convention of the AISB.

Petricek, T., Guerra, G., Syme, D. (2015). *F# Data: Making structured data first-class citizens*. Submitted. Available at: http://tomasp.net/academic/drafts/fsharp-data/

Syme, D. (2011). *F# 3.0: data, services, Web, cloud, at your fingertips (talk)*. Available at: http://channel9.msdn.com/events/BUILD/BUILD2011/SAC-904T

Syme, D., et al. (2013). *Themes in information-rich functional programming for internet-scale data sources*. In Proceedings of DDFP workshop.

Wadler, P. (1998). *The expression problem*. Sent to the Java-genericity mailing list.

Wittgenstein, L. (2009). Philosophical Investigations (4th ed.) Blackwell Publishing Ltd. ISBN: 978-0024288103

Wright, A., Felleisen, M. (1994). *A syntactic approach to type soundness*. J. Inf. Comput. vol. 115, n. 1