Against a universal definition of 'type'

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. There are many definitions used by different communities, but none of them applies universally. A brief excursion into philosophy of science shows that this is not just tolerable, but necessary for progress. Philosophy also suggests how we can think about this imprecise notion of type.

> *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 'type'. Although there are many formal definitions, in practice a 'type' it is often used as 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 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?

If we review academic literature concerning types, we find a number of formal definitions. But each definition comes with a different intuition behind types, different tools for working with then and also different motivation for using types[4]. Thus those looking for a universal definition of a type that would apply to all the uses are determined to fail. The meaning of a type also changes over time (we usually do not notice) and different uses require different properties (that often do not share a common ground).

If we look past 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 we do not need a formal universal definition of a type. I discuss the issue from the perspective of philosophy of science, looking how similar issues have been treated in mathematics, philosophy of language and sciences.

I first discuss how the meaning of types differs between communities and how it changes over time. Then I look for arguments supporting the idea that the notion of 'type' should be left undefined. Finally, I discuss 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] The biggest divide, identified by Kell (2014), is between 'expression types' arising from the logical tradition and 'data types' following an engineering tradition. In this essay, I look at some of the finer distinction within the logical tradition. Even one side of this big divide is surprisingly complex!

How the meaning of types changes

Allow me 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]:

1. Detecting errors via type-checking, statically or dynamically
2. Abstraction and support for structuring large systems
3. Documentation
4. Efficiency
5. Whole-language safety

My students were already indoctrinated and did not question the list[6]. Indeed, past languages used types for *all* of the reasons above. Nowadays, types are used for *some* of the reasons, but rarely all of them. As already mentioned, TypeScript sacrifices safety (5) and efficiency (4) and uses types for documentation (3) and limited compile-time error detection (1). Julia[7] goes further and uses types, but not for rejecting programs[8]. Types are still important for expressiveness (not on our list), system structuring (2) and documentation (3).

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

If we now follow the development of types from Russell to modern programming, we'll find different definitions leading to different intuitions and use cases for types, even if we stay just within the narrow 'logical tradition'.

From foundations of mathematics to the lambda calculus

Types first appeared in Russell's paper *Mathematical logic as based on the theory of types* in 1908. He uses types to avoid paradoxes of the kind "class of all classes that do not contain themselves as elements." Compared with types in programming, Russell's types have quite different definition and uses.

> *A type is defined as the range of significance of a propositional function, i.e., as the collection of arguments for which the said function has values.*[10]

First of all, Russell defines types of propositions based on their inputs (what we would call *domain*). This contrasts with the use in programming where types are often interpreted as the sets of results of an expression, or the *range*. However, what types *are* does not matter to Russell: *"It is unnecessary, in practice, to know what objects belong to the lowest type (...). For in practice, only the relative types of variables are relevant."* The paper does not build on this definition and instead works with a hierarchy of types such that propositions containing variables of type n are assigned type $n + 1$. The theory of types then avoids contradictions arising from self-reference as follows:

> *[W]hen 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.* [11]

Also note that Russell's types do not rule out propositions as invalid. Instead, they *change their meaning* to avoid the contradiction. We can still say "I am lying", but it means a different thing than without types. So, while there is a clear connection between Russell's types and types in programming, it would be a mistake to think that they are really the same.

Russell's types inspired Church's work on λ-calculus. It is important to understand that back then, the λ-calculus was not understood as a simple programming language. It appeared (together with Turing's machines and the theory of recursive functions) 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.

Church's paper is a contribution to the foundations of mathematics. He discusses how to combine the formalism of λ-calculus with the theory of types. However, Church does not elaborate on possible uses of this system. In particular, he does not link the system to Russell's paradoxes and his original paper does not discuss which λ-terms cannot be assigned a type – a crucial use case for programming languages!

Church uses types as a purely formal construct. His system includes two base types; o for propositions and ι for individuals, but he does not define what his types 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.*[14]

Although Church's notion of types is formally close to types in functional programming languages, the intuition behind types was very different. Church did not see types as "sets of possible

[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] A popular slogan is that *Julia uses the type system in all the ways that don't end with the programmer arguing with the compiler*, Hanson (2013).

[10] Russell (1908), 236

[11] Russell (1908), 240

[14] Church (1940)

values", which is a more recent view discussed next. He also does not introduce type systems in order to rule out certain terms. In other words, none of the points from the list at the beginning of this section applies to Church's typed λ-calculus.

From expression types to computation types

The nowadays common interpretation of types as sets of values appeared much later than types themselves. This development is interesting because it is where the logical tradition (types in λ-calculus) meets the engineering tradition (data storage in computers). According to Priestley, the view appeared in early 1970s thanks to Hoare, building on the work of McCarthy:

> *[McCarthy's theory] was further developed by Hoare, who proposed that data types in programming language could be understood as denoting sets of data values.*[15]

The new theory of types influenced early programming languages like Pascal and ML. As Priestley points out, treating types as sets had its issues, for example *"there is no obvious set-theoretic analogue to pointers"*[16].

Many people using programming languages nowadays intuitively see types as sets. This is so fundamental idea that it is hard to unsee. Indeed, we are tempted to interpret the types in Church's simply typed λ-calculus as sets too. However, doing so is a misinterpretation of the original work. This is not a problem for normal science, but it matters when we try to get at the core of what types are. In other words, there is a small subtle change in how we think about types and we might not even notice it if we are not explicitly searching for it!

The subtle change in the meaning of types affects not just what types are, but also what can be done with them. When we see types as sets, it makes sense to prove that evaluating a program of a certain type produces values that belong to the set denoted by this type. This is the key principle behind the syntactic approach to type soundness introduced by Wright and Felleisen[19] and taught in standard textbooks on types. In summary, seeing types in a certain way leads to different intuition behind them (things that do not fit the intuition well will appear in later chapters of our textbooks, if at all) and it also shapes what questions about types can be asked.

However, seeing types as sets of values 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[21]. 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 (read using "!") 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 an 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 becomes difficult. If we ignore the effects, we are leaving out a crucial part of the story. If we attempt to integrate effects into the sets, our sets become extremely complex (a set of numbers turns into a set of functions that take a model of the world and produce an integer together with a new world). At this point, it might be easier to find a different meaning for types that does not lead to such complexity. What we are facing here is akin to Kuhnian *paradigm shift*. When it becomes hard to solve puzzles using the established methods, scientists adopt different definitions and different methods. One such alternative view that lets us talk about effects is to treat types as relations that has been advocated by Benton in his 2014 talk:

> *Express meaning of high-level types as relational, extensional constraints on the behaviour of compiled code*[22]

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 (Benton claims that *"Types are about abstractions not about errors"*) and perhaps more importantly, we also need to change our methods for working with types. For example, the notion of syntactic type safety becomes meaningless.

Dependent types and homotopy type theory

From the practical perspective, dependent types aim to make types more precise. A type of an array might include the size of the array, making it possible to verify the absence of out-of-bounds accesses statically[23]. Here, we can think of types as sets, but again, dependent types go further and allow specifying more complex program properties that (like memory effects), do not fit this view.

More interestingly, dependent types can be also seen as going back to the logic and foundations of mathematics:

[15] Priestley (2011), 246

[16] ibid. 247. We cannot see pointers as sets of addresses, because there is a difference between a pointer to a record and a pointer to an integer. Treating pointers as sets of addresses would now require a model of memory!

[19] Wright, Felleisen (1994)

[21] Lucassen, Gifford (1988)

[22] Benton (2014)

[23] This example follows Chlipala (2014), 8

> *Generalizing the [Curry-Howard] correspondence to first-order predicate logic naturally leads to dependent types.*[24]

Dependent types introduces two notable type constructors: *dependent functions* and *dependent pairs*. Those correspond to universal and existential quantifiers from predicate logic. Both can be interpreted as sets, but again, we soon face issues that are difficult to resolve using the set-based model. This might, in part, be a reason for the recent interest in homotopy type theory, which uses yet another interpretation of types:

> *The central new idea in homotopy type theory is that types can be regarded as spaces in homotopy theory, or higher-dimensional groupoids in category theory.*[26]

Dependently typed programming and homotopy type theory also change what types are good for. Rather than focusing on programming (the list from the beginning of the section), types are now used for theorem proving (through their connection with logic) and for building foundations of mathematics.

Unsound and relatively sounds type systems

So far, we could think that there is an ultimate ideal notion of type that we are slowly getting closer to. However, the following two developments happen in parallel with the one discussed last and they take very different directions.

TypeScript and Dart are two languages that both compile to JavaScript and both have an unsound type system. They are not *unsafe*[27], because the execution engine checks types dynamically. 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 (...)*[28]

The first two 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. In the sense discussed by Hoare and Benton, types in Dart and TypeScript do not "mean anything". Types are still used for (limited) error detection but their main purpose shifts from safety to documentation and tool support. This might be a small step for a programmer, but it is a giant (and unacceptable) leap for a mathematician.

Finally, the third development comes with type providers in F# and Idris[29]. 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 countries as types with indicators as statically checked fields. Does this change what types mean? When we have a type such as "Czech Republic", it is better seen as an individual of an information science ontology[30], then as a set!

Type providers are interesting, because they do not introduce unsoundness *per se* (F# is very strict about types in many ways). However, the soundness of programs becomes relative with respect to some aspects of the external world. A program accessing information about Czech Republic is sound as long as the country does not disappear from the external data source.

Type providers are yet another development of both meaning and purpose of types. Types serve for both error checking (with a relativized twist) and as a documentation for a human and a machine (to provide auto-complete), but at the same time, they require quite different intuition.

A universal definition of type

This incomplete review shows that types are not a single well defined concept. Sometimes, but not always, we can find a precise definition, but none of the definitions can capture all the uses that we find throughout the history of 'types'.

A follower of a certain tradition can choose one definition and extend it so that it covers other uses. But as I attempted to show in this section, if we do so, we miss the point that other users of 'types' consider crucial. We can see types as sets and construct complex sets to model effectful programs, but we do not learn what programs actually do. Or we can treat type representing Czech Republic as a set, but it becomes vacuous and loses important connection with the external world.

As we move between different traditions, the meaning and the purpose of types changes and it is easy to imagine that this will continue for future uses of types. So, what can we do if we want to talk about types and still capture all of their rich and diverse uses? I believe that we can explain many of those developments and find interesting ideas for talking about types by looking at philosophy of science.

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

[26] Univalent Foundations Program (2013), 62

[27] In the usual sense, i.e. that a program could cause unchecked runtime error

[28] Bracha (2011); performance become less important in later work on Dart

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

[30] The work of Leinberger et al. (2014) who implement type provider for semantic web ontologies makes the link with information theory explicit.

Is inconsistent and evolving meaning harmful?

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

Two theories of philosophy of science explore 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 explains how our intuitive understanding of entities evolves.

Inconsistent theories and research programmes

Lakatos's theory of research programmes gives us a perspective that can explain inconsistencies between different definitions of 'type'[32]. In this view, a science consists of 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.*[33]

This theory states 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.

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 three different programmes:

- The textbook definition by Pierce[34] captures the core assumptions of one research programme. We can see types as sets and come with sound, tractable type systems that serve to detect errors. The programme also provides standard tools such as syntactic soundness.
- According to the programme advocated by Benton, types should have a meaning (as relations). 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, but its proponents are willing to sacrifice properties like whole-language safety. The methods include e.g., using types for editor tooling.

Describing the research programmes precisely in detail is work that I leave to the future historians 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.

A similar point has been made by Feyerabend who argues against the *consistency condition*, which requires that scientific theories should be consistent with previous work:

> *[T]he methodological unit to which we must refer [is] a whole set of partly overlapping, factually adequate, but mutually inconsistent theories.*[35]

Are the different definitions of types discussed above mutually inconsistent? I believe so. It is difficult to see how we could talk about logical types from foundations of mathematics and unsound types of Dart at the same time. Yet, it is still useful to think about both of them as an instance of the same concept!

Types as sociological boundary objects

Should we then identify the distinct research programmes and name the concept of type differently and unambiguously in each of them to avoid confusion? 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.*[37]

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 a common name for types across multiple research programmes. As boundary objects, types are very 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.*[38]

[32] Paradigms and paradigm shifts introduced by Kuhn are also related, but they apply to the whole community and so are perhaps less directly applicable here, although some developments resemble paradigm shifts.

[33] Chalmers (1999)

[34] Pierce (2002)

[35] Feyerabend (2010), 20

[37] Star, Griesemer (1989)

[38] Ibid

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

How meaning changes through concept stretching

To understand how the meaning of a type changes, we can find inspiration in philosophy of mathematics. In 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) and describes how mathematicians face numerous counterexamples that were discovered (such as nonconvex polyhedra, polyhedra with tunnels etc.).

Lakatos introduces the notion of *concept stretching*, which happens when a new counterexample (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.*[39]

Concept stretching takes a concept and extends it to include an 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). Type providers are another example. They relativize the notion of safety and also suddenly provide thousands of types (or more) and so some of the established methods for working with types become unsuitable. (As a down-to-earth example, auto-completion lists become so long that they now need a search box!)

In Lakatos's story, there are monster-barrers who try to save the original interpretations and methods by labelling the newly discovered counterexamples 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 (...) delusion that monster-barring narrows concepts, while in fact, it keeps them invariant.*[40]

The fact that concept stretching happens secretly is interesting for our discussion about types too. For example, the shift from Church's simply typed lambda calculus to types in functional languages is larger than generally understood. However, once we see types as sets of values, it is very hard to go back and see the world through Church's original perspective.

The introduction of unsound type systems is another example of concept stretching. Like adding a tunnel through a polyhedra, it extends the concept of a type 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.*

Should we be precise about types?

Research programmes and concept stretching help us better track how types are used. 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.

Doing this is, indeed, a useful contribution to science, but it can only be done in retrospect once we know all the facts. As noted by Latour in Science in Action[41], there are two sides: on the left, we know all the facts and have many strong allies; on the right, everything is in the making and under-determined. The work on the right is not a black-boxed science (yet), but once it becomes a black-box, it is as solid as anything else.

This explains why we cannot point a finger at interesting work that has been unjustly rejected, e.g. for the lack of formalism. The things on the right side are not science, because they are not science[42]! Ubiquitous focus on formalism does not rule out parts of science. It defines what a science is.

Against the definition of type

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:

[39] Lakatos (1979), 84

[40] Lakatos (1979), 86

[41] Latour (1987)

[42] To avoid the tautology, just imagine that the statement on the left talks about time t and the statement on the right talks about time $t - 1$.

> *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 spirit of their students with this dumb ritual of analysis, this blind, rote, eternal naming of things.*[43]

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's philosophy presents a similar idea, but more seriously and with historical grounding.

Epistemological anarchism and clarity of terms

Searching for clarity is worthwhile, 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.*[44]

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. One could invent an inelegant answer, but this would shift the focus of the work in a 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.*[45]

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*[46].

How science actually works

Feyerabend's position may be extreme for some readers, but he is not alone. 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*[47]

> *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*[48]

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 possible to see in retrospect, but not during the development itself. Feyerabend summarizes this position with his famous slogan:

> *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*

[43] Pirsig (1999), 360
44 Feyerabend (2010), 200
45 Ibid, 200
46 Feyerabend (2010)
47 Chalmers (1999), 135
48 Chalmers (1999), 106

> *be defended under all circumstances and in all stages of human development. It is the principle: anything goes.*[49]

As Feyerabend later said, 'anything goes' is not a principle, but *the terrified exclamation of a rationalist who takes a closer look at history*[50]. 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.*[51]

Looking at the history of science supports the main idea of this essay. That is, we should not require a precise definition of the notion of 'type'. 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 are two areas where new ways of thinking about types can be especially valuable. The first is in new and previously unexplored domains. When types are used in a new domain, their meaning might change and it can take time before we settle on a clear formal definition. The second area is when we want to talk about types universally and include many of the rich and diverse precise 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. That these provide a useful complement to the rigorous methods that we use when operating within a narrow and formalized areas of an established research programme.

There are many theories to look at, but in this essay, I explore 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.*[52]

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.*[53]

Similarly, the meaning of a scientific term can be explained by its use in the scientific community. When discussing different notions of types earlier, we looked at both what types are (undefined hierarchy, sets, spaces), but also how they are used (avoiding errors, providing documentation, etc.).

The idea here is that we focus just on how types are used, because this is what types *are*. This may sound unorthodox, but it resolves one of the key issues we face when looking for a universal notion of type – studying the use does not require consistent definitions.

To understand types, we can study how they are used in different contexts. Wittgenstein calls these contexts *language games*, but what are the language games surrounding types? There are natural contexts that already exist and there are 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, but

[49] Feyerabend (2010), 12
[50] Ibid, vii
[51] Ibid, 160
[52] Feyerabend (2010), 200
[53] Wittgenstein (2009), no.43

has only regained prominence with the later developments of Per Martin-Löf's type theory and homotopy type theory.

However, documenting 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 new 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*[54]. The problem is extending a set of objects and functions in two directions – by adding new kinds of objects and new functions.

For example, objects may represent numerical expressions (constant, variable, addition) and functions operations over them (pretty printing, evaluation). In some type systems, it is easy to add new kinds of objects. In other type systems, we can add functions, but adding new objects is hard.

The language game sets perhaps unreasonable constraints (we should not require recompilation), but that is not a flaw. Instead, it reveals the abstraction and error-checking capabilities of a system. At the same time, it can be 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, and so constructing language games to explore other properties of types is one interesting open question of this essay.

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.*[55]

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 (...).*[56]

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. A type is a decidable syntactic program property, but a type that cannot be effectively decided is still a type. A type can be used to rule out errors, but a type that does not rule out all errors is still a type.

Another useful point made by Hacking is that illustrations in children books illustrate the stripiness of tigers to build the stereotype. Similarly, the Types lecture notes at the start of this essay and computer science textbooks discuss properties of

[54] Wadler (1998)

[55] Hacking (1983), 75

[56] Putnam, p230

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.*[57]

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.

We should also keep stereotypes in mind when reading textbooks. A textbook description is two things – a formal definition 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, a similar point has been made by Richard P. Gabriel in his recent 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.*[58]

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.*[59]

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*[60].

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.*[61]

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, but without a full theory that would be usual in theory-founded work. Yet, type providers already influenced other languages[63] and the theory explaining them started appearing too.[64]

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

[57] Putnam (1979), 250
[58] Gabriel (2012)
[59] Hacking (1983), 158
[60] Ibid, 175
[61] Ibid, 150
[63] Christiansen (2013)
[64] Petricek (2015)

Types as scientific entities and types in practice

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.*[65]

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. Tools such as theorem provers are an interesting example. They have types at their very core, but are used to create other valuable things using them.

The other option is to 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[66], but I believe that this is an underexplored area with interesting possibilities. For example, the Future of Programming workshop[67] 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 (...).*[68]

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[69]. 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. The webcast format pioneered at the Future of Programming workshop is a good example.

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[70]:

> *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 (...).*[71]

Conclusions

This essay was inspired by the frequent misunderstandings when discussing types. Although we have good understanding of types in narrow domains, I argued that it is impossible to give a formal and universal definition of what a type is. 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.

To motivate the essay, I started with a brief and incomplete history of types. As the examples demonstrate, the meaning

[65] Hacking (1983), 262

[66] Petricek (2014)

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

[68] Chalmers (1999), 194

[69] 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.

[70] Hacking (1983), xiii

[71] Hacking (1983), 154

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 and requiring such precision can even harm scientific progress.

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/

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.

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

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

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.

Hanson, L (2013). Quoted in *Function Argument Intent*, julia-users mailing list: https://groups.google.com/forum/#!msg/julia-users/qJa2EyUXJfo/W3bfHeXhuHEJ

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.

Latour, B. (1987). Science in Action. Harvard University Press. ISBN 978-0-674-79291-3

Leinberger, M., et al. (2014). *Semantic Web Application development with LITEQ*. Proceedings of ISWC, Springer.

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

Pierce, B. C., (2002). *Types and Programming Languages*. MIT Press, ISBN 0-262-16209-1

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.

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

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

Russell, B. (1908). *Mathematical logic as based on the Theory of Types*. American Journal of Mathematics, Vol.30, No.3, 222-262.

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

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

Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. Available at: http://homotopytypetheory.org/book

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

Wittgenstein, L. (2009). Philosophical Investigations (4^{th} 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