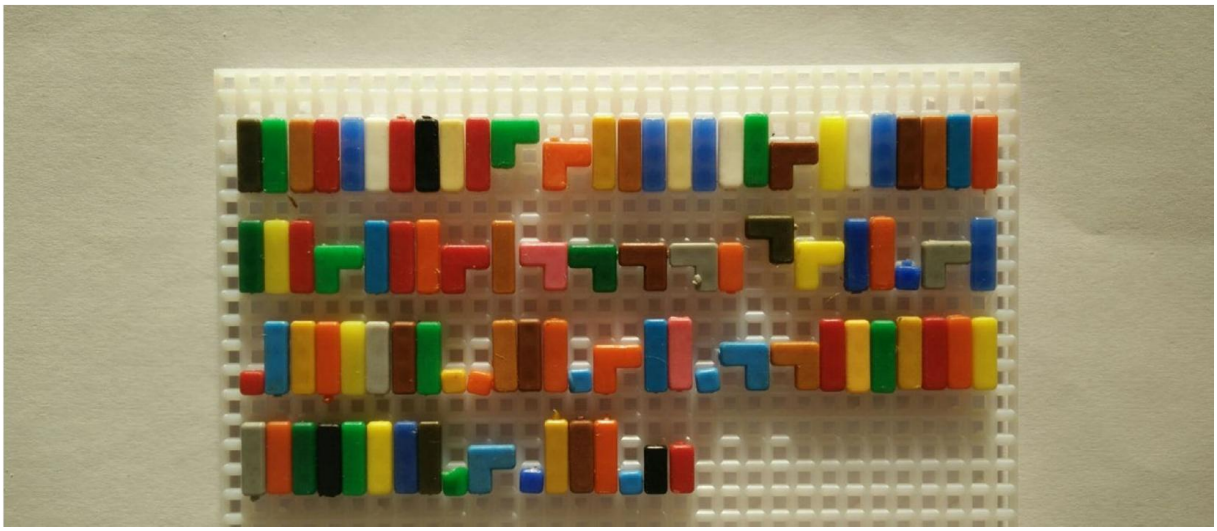
Understanding computer science

How the discipline of computer science develops new understanding

Master Thesis Philosophy of Science, Technology and Society

Joke Noppers



University of Twente, Faculty of Behavioral, Management and Social Sciences, Enschede, the Netherlands

August, 31, 2017

Graduation committee

First reader: Fokko Jan Dijksterhuis

Second reader: Miles MacLeod

Contents

Chapter 0: preface

The nature of computers and computer science has always puzzled me. Somehow, the emergence of computers and computer science has opened up a new perspective, a new way of thinking that enabled us to profoundly change the world we live in. But what *kind* of perspective did computer science open up? In my final thesis for the master's program of Philosophy of Science, Technology and Society, I seek to understand this new kind of thinking.

Writing this thesis has certainly not been an easy process. While the philosophy of science profoundly analyzed the more traditional forms of scientific thinking, computer science still remains a philosophical *terra incognita*. And it is difficult to ask the right questions when you have no concepts, theories or frameworks to formulate your question in.

But, to paraphrase President Kennedy, sometimes, you don't do things because they are easy. You do them because they are *hard.* This conceptual struggle made my job very difficult. But it also made my job deeply challenging and satisfying. Before finishing my thesis, I wouldn't have imagined that spending all those days behind my desktop computer could be so much fun. But it was.

I am grateful to my thesis supervisor, Fokko Jan Dijksterhuis, for his guidance, his feedback and for those many fun and energizing meetings in which we discussed my progress. I am also grateful to Mieke Boon, who helped me a lot during the beginning of my thesis. I would like to thank Miles MacLeod, who was willing to step in as my second supervisor. And I am grateful to Yvonne Luyten-de Thouars, who motivated me to finish my thesis at a moment when life's priorities had shifted towards my other work.

I would like to thank my boyfriend, Ivo Nouwens, for providing the emotional and practical support that enabled me to finish my thesis. I am also grateful for Lantz Miller and the students from my Masterlab graduation group, who were my sparring partners in the

writing process. I would like to thank Arthur Melissen, for providing feedback and advice on my thesis. I am grateful to my friends and family, for supporting me. And last but not least, I am grateful to my computer and my smartphone, the amazing products of seven decades of computing, who were indispensable to the writing process of this thesis.

Joke Noppers

Enschede, August 28, 2017

Chapter 1: Understanding in computer science

Introduction

Science helps us understand the world we live in. It makes us understand the phenomena that happen around us. Think of lightning, the oceanic tides, magnetism, but also things like human social behavior, historic events or economic cycles.

For most disciplines of science, we have a more or less clear idea of how they work to provide understanding. The physicist performs experiments in a lab, to arrive at new insights about our universe. The biologist observes plants and animals, to better understand the natural world. The mathematician works out abstract mathematical theories on a chalk board.

But not all disciplines of science are well-understood. As a scientific discipline, the field of computer science is the source of much confusion. Natural science studies the world around us, to develop new insights about our universe. But what new insights are there to be found in a man-made device like the computer? Some people say that computer science is an abstract field, like mathematics. They claim that it is not really about computers, but about something else. But if that is the case, why do all the topics in this field, directly or indirectly, have something to do with computers?[1]

Apparently, it is unclear what kind of thing computer scientists seeks to understand and how they seek to understand that thing. Because of this, we find it hard to understand how computer science 'works' as a science. This creates a lot of confusion. Even among computer scientists themselves, there is a lot of disagreement about 'what' computer science actually is.

Some people have argued that computer science is an applied science or engineering (Loui, 1995). Some think it is a new branch of mathematics (Dijkstra, 1978; Knuth, 1974), but others have contested it and call it an empirical or natural science (Denning, 2007; Eden,

[1] For an interesting blogpost about this issue, see: http://www.nomachetejuggling.com/2012/02/02/computer-science-and-telescopes/

2007; Newell & Simon, 1976). Some think that it is revolutionary new entirely (Hartmanis, 1995). Others are convinced of the exact opposite: They think that computer science is not even a field at all, but "a grab bag of tenuously related areas, thrown together by an accident of history" (Graham, 2008) .

Perhaps, it is not so surprising that computer science is so poorly understood. When we want to understand the practices of scientific fields, we turn to the philosophy of science. But philosophy of science spent very little attention on analyzing computer science as a scientific practice.

The developments in computing and computer science have attracted a great deal of philosophical interest. For instance, people discussed the meaning of computational processes (Searle, 1980) and the social consequences of information technology (Brey, Light, & Smith, 1998). But there was little attention for understanding the nature of computer *science* (Brey & Søraker, 2009).

From a philosophy of science perspective, computer science is still uncharted territory. The philosophy of science is mainly concerned with the study of established fields, like physics, chemistry, biology, sociology, or economics. There is a focus on empirical, natural science (Ladyman, 2002) and a substantial body of literature about the philosophy mathematics (Horsten, 2016), but not a lot about computer science.

Therefore, I believe that a careful philosophical analysis of the practices of this field would be the first step towards a better understanding of the field. That is what I am trying to achieve in this thesis.

Research question

In my thesis, I want to develop a better understanding of computer science as a science. I want to explain how this field works as a scientific discipline, by showing how computer scientists can develop new scientific understanding.

Research question: How can the field of computer science develop new understanding?

Sub-question 1: What kind of thing does computer science seek to understand?

Sub-question 2: How does computer science develop an understanding of this thing?

With my thesis, I do not aim to provide the definitive conclusion about the status of computer science. I believe that my thesis will contribute to this discussion in a different way. Through this analysis, I will explain what scientific understanding is, how it is developed and how it relates to the practices of a computer science. Therefore, I will develop a clear set of concepts for thinking about understanding in computer science. I hope that these concepts can serve as a set of 'thinking tools', enabling others to formulate their own standpoints in the discussion about the nature of computer science.

My approach

I plan to explain the development of understanding in computer science by relating it to other, more familiar forms of scientific understanding. This approach enables me to discuss an 'unfamiliar' kind of understanding in familiar terms. This, in turn, allows me to explain more clearly how understanding in computer science works.

Because computers are a highly technical subject, the field of computer science is often associated with technical disciplines, such as natural science, mathematics and engineering. Therefore, *if* computer science is a scientific field, most people would likely group it with the 'hard', technical sciences. Therefore, I will relate the understanding developed in computer science to the understanding developed in those fields.

In the natural sciences, two important, different ways of understanding can be discerned. The first kind is an understanding of the physical, developed through experiment and testing. The second kind is mathematical understanding, developed through abstract, mathematical reasoning. I will relate understanding in computer science to these familiar forms of understanding. This will enable me to explain the development of understanding in computer science in familiar terms.

Outline of my thesis

I will use the first part of my thesis to discuss scientific understanding. I will start my thesis with a chapter about scientific understanding in general. In this chapter, I will discuss several concepts of scientific understanding. I will explain how understanding is developed in science and I will show how scientific theories can bring about such understanding. I will use this chapter to define a clear concept of scientific understanding.

When I have defined my concept of scientific understanding, I will use that concept to describe the two important forms of understanding in natural science. First, I will describe how scientists can develop an understanding of the physical world. The field of physics is often seen as the 'model science' for natural science. This field seeks to understand the physical world at its most fundamental level. Therefore, I will use the practices of this field as an example of scientific understanding of the physical world. Then, I will describe the second important form of understanding, which is mathematical understanding. I will show how the field of mathematics develops new understanding, by studying abstract, mathematical concepts.

After having described these two forms of understanding, I will move on to the second part of my thesis. In this part, I will describe and analyze the scientific understanding developed in computer science. I will start with a chapter that explains the methodology of this description. I will discuss the method I have chosen to describe this form of understanding and I will explain why I have chosen this method.

In the next chapter, I will use this method to describe the understanding developed in computer science. I will show how several historical developments changed our understanding of computers. As I will show, this new idea of computing provided the basis for the development of understanding in computer science.

In the next chapter, I will relate this new form of understanding to the other forms of understanding. I will explain what kind of understanding this is, how it is being developed and to what extent

it is different from the understanding developed in mathematics and physics. In the last chapter, I will discuss what this means for my research question and I will give recommendations for further research.

References

Brey, P., Light, A., & Smith, J. M. (1998). Space-shaping technologies and the geographical disembedding of place. *Philosophies of place*, 239-263.

Brey, P., & Søraker, J. H. (2009). Philosophy of computing and information technology. *Philosophy of technology and engineering sciences, 9*, 1341-1408. http://www.idt.mdh.se/kurser/comphil/2011/PHILO-INFORM-TECHNO-20081023.pdf

Denning, P. J. (2007). Computing is a natural science *Communications of the ACM* (Vol. 50, pp. 13-18).

Dijkstra, E. W. (1978). EWD682. The nature of Computer Science (first draft). Edsger W. Dijkstra Archive https://www.cs.utexas.edu/users/EWD/ewd06xx/EWD682.PDF

Eden, A. H. (2007). Three paradigms of computer science. *Minds and Machines, 17*(2), 135-167. doi:10.1007/s11023-007-9060-8

Graham, P. (2004). *Hackers & painters: big ideas from the computer age*. Sebastopol, CA: O'Reilly Media Inc.

Hartmanis, J. (1995). On computational complexity and the nature of computer science. *ACM Computing Surveys (CSUR), 27*(1), 7-16.

Horsten, L. (2016, Winter 2016). Philosophy of Mathematics. *The Stanford Encyclopedia of Philosophy*. Retrieved August 19, 2017, from https://plato.stanford.edu/archives/win2016/entries/philosophy-mathematics/

Knuth, D. E. (1974). Computer science and its relation to mathematics. *The American Mathematical Monthly*. Retrieved from http://www.jstor.org/stable/2318994

Ladyman, J. (2002). *Understanding philosophy of science*. Abingdon: Routledge.

Loui, M. C. (1995). Computer science is a new engineering discipline. *ACM Computing Surveys (CSUR), 27*(1), 31-32. doi:10.1145/214037.214049

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM, 19*(3), 113-126. doi:10.1145/360018.360022

Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and brain sciences, 3*(3), 417-424.

Chapter 2: Developing understanding in science

In the next chapter, I will review the literature in the philosophy of science, in order to find an accurate description of the development of understanding in science. I will use this account to discuss the development of understanding in mathematics and physics and relate this to the development of understanding in computer science.

Understanding in the philosophy of science

Understanding is central to the activity of doing science. Therefore, one would expect that the philosophy of science has spent a great deal of attention at clarifying understanding. Strangely enough, however, this does not seem to be the case.

A lot of work is focused on scientific explanation. These philosophers seek to describe how theories explain things to us. Understanding is mostly treated as a by-product of such scientific explanation. A lot of scholars assume that understanding is the result of a clear and accurate explanation.

For instance, Hempel (1965) conceives of an explanation as a logically valid argument, in which the phenomenon to be explained is deduced from one or more universal laws. The feeling of understanding is no part of the logical chain that connects phenomena with the matters explaining them. For Hempel, this feeling of understanding is only a psychological, subjective byproduct of possessing an explanation. Therefore, understanding is of no interest to philosophers of science, while explanation is. Trout (2002) even argues that focusing on understanding is dangerous, because the feeling of understanding is subjective: People can feel they have understood something, while in reality they do not possess an understanding.

Other philosophers of science point out that a clear explanation does not always bring about understanding. Often, an explanation only leads to a sense of understanding in some people, leaving others puzzled. So understanding is an active psychological process, not a passive by-product of having taken in the right explanation. Personal factors also determine whether a given explanation leads to

understanding or not. Therefore, explanations in themselves do not suffice as a good account of scientific understanding.

Michael Scriven (1962) argues that these personal factors must be taken into account also. Scriven also points out that understanding is not as subjective as Hempel and Trout think it is. Understanding can be tested in a more or less objective manner, as is being done in school examinations. Michael Friedman (1974) also criticizes theories that solely focus on explanations. Friedman argues that these theories try to define a concept of explanation, but they do not explain what it is about this particular concept, that brings about understanding.

Therefore, if I want an accurate and useful description of scientific understanding, I need to move beyond philosophical accounts that focus on scientific explanations.

De Regt: Understanding as use

Henk de Regt (2009) tries to give an account of what understanding is and how it is related to theories and explanation. According to de Regt, an explanation brings about an understanding of a thing if it enables you to reason about that thing.

According to de Regt, a thing T can be understood if a theoretical explanation for it exists that is *intelligible*. Many general theories also explain specific things. They do this by explaining a general mechanism or regularity that is underlying these specific cases. When such a theoretical explanation is intelligible to you, you can use your understanding of the general mechanism to construct your own explanation of specific thing T.

The theory brings about an understanding of things because it enables us to reason about those things, constructing our own explanations of those things. To have an understanding of a thing, then, is being able to reason about that thing.

De Regt's account provides an idea of what understanding means and when it is present. But it does not tell us how theoretical explanations bring about such understanding. What, exactly, makes a

theory 'intelligible?' And how exactly, do theories enable us to reason about things?

Boon: Understanding as interpretation

Mieke Boon's (2009) work on scientific understanding is specifically aimed at this 'how' question. She explains how we can understand things and how theories bring about understanding.

Like de Regt, Boon thinks that to have an understanding of a thing is to be able to reason about it. According to Boon, theoretical explanations provide understanding because they provide something necessary for such reasoning: a set of concepts and relations, which can be used for building conceptual models.

According to Boon, we grasp the fuzzy and confusing reality around us through a process of active interpretation. We divide reality into different concepts and draw relations between those concepts. From these concepts and relations, we build the mental structures that allow us to make sense of the world. These structures form mental models of the things in the world. The concepts and relationships of the model can be used to make inferences about the thing it is representing. Therefore, these structures allow us to reason about those things. Boon calls such a conceptual model an *interpretative structure*.

Boon equals having an understanding of a thing with being able to reason about it. Because these structures are necessary for such reasoning, having an interpretative structure of a thing is a necessary condition for understanding it. According to Boon, theories provide us with an understanding of things by enabling us to build an interpretative structure of those things.

Theories are conceptual frameworks, consisting of a description of certain concepts and the relations that exist between them. These concepts and relations can be used, for structuring and interpreting things in terms of the theory. By structuring and interpreting a thing with these concepts and relations, we build an interpretative structure of it, in the form of a *theoretical model*. This allows us to reason about the thing and therefore, to understand it.

According to de Regt, when a general theory is intelligible to you, you can use your understanding of this general theory to create your own explanations for specific things. Boon's account explains exactly *how* a theory allows you to create your own explanations.

If a theory is intelligible to you, you understand its framework of concepts and relations. Therefore, you can use these concepts and relations for creating interpretative structures of a specific thing. This interpretative structure enables you to understand this thing. This is how an intelligible general theory enables you to create your own explanations of specific things.

When a general theory enables the understanding a specific thing by providing a set of concepts and relations, this theory serves as an *interpretative framework* for understanding that specific thing. By using their understanding as a basis for making further inferences, scientists can refine this understanding. This is how understanding is gradually developed in science.

Understanding in different fields

In the previous sections, I have reviewed the literature in the philosophy of science, in order to find an accurate description of the development of understanding in science. Although most of the literature focuses on scientific explanations, there were two accounts that discussed scientific understanding. The account of de Regt (2009) explains what understanding is, while the account of Boon (2009) shows how the explanations from scientific theories can bring such understanding about.

I will use their accounts to analyze and describe the development of understanding in computer science, mathematics and physics. For each field, I will analyze how they seek to understand the topic that they study.

In the next few chapters, I will discuss what sorts of explanations the field provides, how these explanations enable the understanding of specific things, how the topic of study is to be understood, how this idea is reflected in the field's concept of a theory and how this specific form of understanding affords reasoning about that topic. These detailed accounts of understanding in different fields

will allow me to relate the understanding developed in computer science to understanding in other fields.

References

Boon, M. (2009). Understanding in the Engineering Sciences: Interpretative Structures. In H. W. de Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding: Philosophical Perspectives*. Pittsburgh: The University of Pittsburg Press.

de Regt, H. W. (2009). Understanding and Scientific Explanation. In H. W. d. Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding, Philosophical Perspectives* Pittsburgh: University of Pittsburgh Press.

Friedman, M. (1974). Explanation and Scientific Understanding. *Journal of Philosophy, 71*(1), 5-19.

Hempel, C. (1965). *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*. New York: The Free Press.

Scriven, M. (1962). Explanation, Predictions and Laws. In H. Feigl & G. Maxwell (Eds.), *Scientific Explanation, Space and Time* (Vol. III). Minneapolis: University of Minnesota Press.

Trout, J. (2002). Scientific Explanation And The Sense Of Understanding. *Philosophy of Science, 69*(2), 212-233.

Chapter 3: The development of understanding in physics

In the previous chapter, I used the work of Boon (2009) and de Regt (2009) to develop a general description of understanding in science. In this chapter, I will use their account of scientific understanding to describe the development of understanding in physics.

I will begin this chapter with a discussion of the specific sort of explanation that physics seeks. I will show how these explanations lead to new understanding. Then, I will discuss what kind of understanding this is. I will discuss how this particular kind of understanding is reflected in the practices of the field. Next, I will discuss the specific assumptions that underlie this way of seeking understanding. I will conclude this chapter with a short overview of the philosophical debate about understanding physical reality.

The specific sort of explanation that physics seeks

Physics is the study of our material universe. The field concerns itself with the behavior of physical matter, which is causing the physical interactions and physical phenomena we observe around us. As I will show, physics understands this physical world in a specific manner. In the next sections, I will discuss what kind of explanations physics seeks how these explanations lead to understanding and how such an understanding helps physicists to make sense of the world.

Physics wants to do more than describe what the world is like. It wants to explain *why* it is like that. Why do physical phenomena happen? Why do objects tend to fall towards the Earth? Why do certain elements emit radiation? Why does salt dissolve in water?

The explanations physics seeks are general explanations. Physics is not interested in explaining why any particular object happened to fall to Earth. The field seeks to explain why objects in general tend to do this. Or, to put it in even more general terms, it wants to know why objects with mass attract each other.

Causes as a key to explaining the physical world

Physics seeks explanations in terms of causes. In natural science, to explain a physical phenomenon is to understand what caused it. There is a good reason why scientists want to understand the causes of physical phenomena. They are the key to understanding our physical world. If we understand what is causing the things happening around us, we can reason about those things, connect different things through a common cause, predict them and in many cases, influence them. This understanding helps natural scientists to grasp the workings of material universe we live in.

Physics is interested in a specific kind of causal explanation. In daily life, the cause of a physical event is often taken to be another physical event. For instance, people might say that a forest fire is caused by a lightning strike. Physics however, looks for a different sort of cause.

This is does not mean that physicists are somehow denying that lightning strikes can induce forest fires. They have very good reasons to believe that the one somehow induces the other. But to natural science, the occurrence of such a lightning strike alone does not fully explain the causes of the fire.

Experience has taught us that lightning strikes can induce forest fires. But our experience does not teach us *why* the event of a lightning strike results in a forest fire instead of green flashes of light, or a glitter explosion. Nor does it teach us *what* it is, that causes lightning strikes to set forests on fire. Therefore, to the physicist, a preceding physical event, like a lightning strike, does not provide a full explanation for the occurrence of a forest fire.

Therefore, physics seeks the causes somewhere else. Physicists believe that more complete explanation for physical phenomena can be found in the behavior of matter. The specific ways in which matter behaves determines the course physical events will take. Therefore, these behavior patterns are seen as the real reason that things happen the way they do. This means that understanding those patterns is crucial for developing an understanding of the physical world.

Forest fires happen because the matter in forests and lightning bolts is behaving in a particular way, causing the lightning strike to set into motion a course of events that will result in a forest fire. Therefore, to understand why the fire occurred, physicists need to understand why the matter in the forest and the lightning bolt behaves the way it does.

Physics does not seek an explanation for physical phenomena in the occurrence of other physical phenomena. This is because the occurrence of these events does not tell physicists very much. For them, understanding the behavior of matter is the key to understanding the causes of physical phenomena.

Explanation beyond matter

But an explanation for the behavior of matter cannot be found within matter itself. Physical things behave in a certain way. But we are not able to observe the causes of this behavior. Such causes cannot be detected through any physical means. Therefore, physics must assume that the behavior of matter is caused by things outside the realm of the visible. These hidden causes are taken to govern the world of visible things, but they are not visible things themselves.

One example of such a hidden cause is gravity. Gravity may not appear 'hidden' to us, because we seem to observe it all the time. But we never observe gravity itself. All we see are the effects it has on the behavior of matter. Gravity wave detectors like VIRGO and LIGO do not observe actual gravity waves. What they measure are slight oscillations in the measuring equipment that are taken to result from these waves. The effects are visible everywhere, but the cause remains hidden.

Electromagnetism, the cause behind visible light is invisible as well. We think we can see it, but the visible lights we observe are not electromagnetism itself. They are a *result* of electromagnetism. This hidden force causes material particles to act in certain way. And when some of these particles reach our retinas, we observe visible light. But the electromagnetic force behind the phenomenon of light is never observed.

While physics seems to be a study of the visible, physical world, the actual focus of the field is with the invisible. Physics seeks an explanation for physical phenomena in a set of hidden causes, because this explanation cannot be found in visible matter itself. This may seem a bit contradictory to our ideas about natural science, because physics seems to be explaining phenomena from basic matter all the time.

For instance, the tendency of chemical elements to bond with other elements is said to be caused by the basic structure of their atoms. Some of those elements have empty spots in their electron shells. 'Sharing' their electrons with other elements can help them fill those spots. Therefore, elements that 'complete' each other tend to form chemical bonds. Apparently, the behavior of these elements is caused by their material composition. This would imply that the explanation for the behavior of matter *can* be found in the matter itself.

However, this explanation merely involves the basic structure of the atom. It does not follow from it. It would be impossible to derive such an explanation from the material structure of the atom alone. This is because the behavior of the atom is not fully determined by this material shape. The atom may possess a number of electron shells, which may or may not have empty spots in them. But this says nothing about a possible tendency of the atom to do something with these electron shells, such as wanting to keep them either full or empty.

Therefore, the basic structure of the atom tells us very little about the way it behaves. Because of this, physics has to seek an explanation beyond the visible. The invisible force of electromagnetism is taken to determine the atom's 'preference' for a full electron shell.

This example nicely illustrates that, even in stories about the fundamental building blocks of matter, a full explanation is to be sought beyond visible matter. Without the intangible force of electromagnetism, the statement that atoms form bonds because their shells aren't full makes as little sense as the statement that

people move to Belgium because they have an uneven number of ping pong balls in their pockets.

How these specific explanations enable understanding

I have shown how physics seeks to explain physical phenomena in terms of hidden causes. Now, I will discuss how such explanations enable the development of an understanding of the physical world. I will show that such explanations bring about understanding by providing 'the missing pieces', completing our picture of the physical world.

From the perspective of physics, our view of the physical world is incomplete at best. In this field, physical phenomena are taken to be the result of hidden causes. This means, that if we observe a physical phenomenon, we only observe half of it. We can see the visible part, but the causes that are driving this phenomenon remain hidden to us.

This limited, partial view of the physical world prevents us from understanding the phenomenon. As explained by Boon (2009) understanding something requires us to have an interpretative structure of the thing. Such an interpretative structure is a model, embodying our ideas of what the thing is like. This structure allows us to reason about the thing and to understand how it works. If we do not have a complete picture of something we cannot create an interpretative structure of it. Therefore, we cannot have an understanding of it.

Explanations in physics help us to complete our picture of those phenomena. They do this by giving an account of the hidden causes.

Theories in physics often describe these hidden causes in general terms. But, as Boon already explained these general descriptions can be used to understand more specific matters also. They can serve as *interpretative frameworks*. The concepts and relations from the general theory are used to create an interpretative structure of the specific thing, enabling scientists to understand that specific thing.

In natural science, general theories can serve as interpretative frameworks for specific phenomena, because they describe the causes of these phenomena. If a specific phenomenon P would be caused by a general, hidden cause C, then a description of general cause C would also be describing the hidden cause behind P. That would mean that a general theory describing C can be used to create a description of the hidden causes behind P.

Such a description of P's invisible causes provides physicists with the missing piece they need to complete their picture of P. They can combine this account with the visible aspects of P that they were already familiar with. By doing this, they create a complete model of P, describing both the visible phenomenon and its hidden causes. Such a complete model of P allows for reasoning about P.

By making inferences about their model of P, scientists can make inferences about P itself. The interpretative structure of P allows them to explain past occurrences of P and to predict future ones. They can use this structure to make connections between P and models of other phenomena and hidden causes. It allows them to think about P, to predict it and to control it to some degree. It gives them an understanding of P they can work with.

The kind of understanding that is being developed

In the previous sections, I have explained where understanding in physics comes from, by showing how this field seeks to explain and how such explanations can bring about understanding. Now, I want to discuss the nature of this understanding itself.

As I have argued an explanation from physics enables us to understand a natural phenomenon, which, in turn, allows us to reason about this phenomenon. But what does it mean to understand a phenomenon through understanding its hidden causes? What sort of things do you understand when you understand a phenomenon in this way? How exactly, can you use such an understanding for making inferences? And what kind of inferences are those? In the next sections, I will show that the specific explanations from physics lead to a specific sort of understanding, which in turn, affords a specific kind of inference.

Understanding in physics: understanding by proxy

As I have shown, theories in physics enable us to understand a phenomenon P by telling a story about a possible hidden cause of P. This hidden cause provides the missing piece we need to complete our model of P. Such a complete model, incorporating both the visible and the invisible aspects of P enables us to reason about P. Therefore, the theory has provided us with an understanding of P.

However, these theories do not complete our understanding of P by providing the *real* missing piece. They merely fill in the blanks by providing *ideas* of what such a missing piece could look like. This is because they cannot provide an account of P's actual hidden cause. An understanding of P's actual hidden causes is unattainable to us, because these causes are taken to be fundamentally inaccessible to us.

Since these causes are taken to be invisible, they cannot be directly observed by us. These causes also do not seem to follow from logical necessity. We can use logic to deduce that a statement like 'In my street, it is either raining or not raining" must necessarily be true. But it is very hard to show that logic dictates that gravity must necessarily exist, or that the speed of light must necessarily be 299,792,458 meters per second. No one, not even the greatest minds in history, has even come close to successfully providing such an argument. Therefore, it is most likely that the nature of these causes cannot be deduced by logic alone.

Because these causes are invisible to us and because we cannot deduce their nature from logic, they are inaccessible to us. We can only know them in an indirect manner. By observing visible phenomena, we can make inferences about their possible, underlying causes. But the real nature of their causes will always remain a secret to us. Therefore, these hidden causes can never be directly understood by us.

Since we cannot have a direct understanding of these hidden causes, physics must provide an alternative for this. The field does this by building theoretical models of the hidden causes. Those models provide a proxy for understanding the real hidden causes. By

understanding those models and reasoning about them, we indirectly reason about the hidden causes they represent. These models enable us to make indirect inferences about these hidden causes. Therefore, they replace the direct understanding that we lack.

This means that, in physics, we never understand the actual hidden causes. These are taken to be inaccessible to us. Instead, we understand a theoretical model of those causes, based on *our ideas* of these causes. Therefore, to have an understanding of P's hidden causes actually means: to have an understanding of our ideas of P's hidden causes. Understanding in physics is an understanding by proxy.

How the nature of this understanding is reflected in the theories of the field

The indirectness of the understanding created by physics is reflected in its concept of a theory. As I have shown, physics explains physical phenomena by providing a cause for these phenomena. But because such causes are invisible to us and cannot be deduced trough logic, we cannot have a direct understanding of them. This means that we cannot directly describe this reality. We can only make indirect inferences about it.

Therefore, in physics, theories are not descriptions of that which is the case. Instead, theories and theoretical models express ideas of what *could* be the case. This means that the concept of a theory in physics is very close to the notion of a theory as it is used in daily speech. They are beliefs about the nature of an unknown reality that is arrived at by evidence and logical inference.

The inferences that this understanding affords

Physics provides a specific sort of explanation for physical phenomena, which leads to a specific, indirect kind of understanding. This in turn, affords a specific kind of reasoning about these phenomena. The practice of developing understanding in physics is based on this specific way of reasoning.

As I have explained, physics understands physical phenomena in an indirect way. The understanding of a model of phenomenon P serves as a proxy for the unattainable, direct understanding of the actual P.

This model is used for making inferences and predictions about the behavior of P.

Because an understanding of the actual P is impossible, scientists cannot know whether their model of P matches the actual P. It could very well be that the actual P is different from the model of P. And such a difference could result in different behavior. To be useful for making inferences and predictions about P, the predictions of a model must match the behavior of the actual P. Therefore, physicists have to check the predictions from their model against the behavior of the real P. This makes physics an empirical science.

If the predictions of the model do not match the behavior of the real P, it means that the model is not useful for reasoning about the real P. Then, the model needs to be revised. If the real behavior of P does match the predictions from the model, it means that the model is useful for predicting the behavior of the real P. Therefore, according to this specific test, the model does not have to be revised.

Such a match does *not* imply that the real P matches the model, however. We can only observe that the material behavior of P matches the behavior predicted by the model. We can never observe whether the hidden causes of P match the ideas expressed in the model. It could be the case that P's underlying causes are totally different from the model, but happen to result in the same behavior as the model predicts.

For instance, the Newtonian model of classical mechanics predicts the motion of physical bodies very well. Only at very high speeds, near the velocity of light, the predictions from this model begin to fall apart, indicating that underlying causes of the motion of these bodies are different from the causes described by Newton's model.

How such inferences lead to the further development of that understanding

Such empirical tests are more than validity checks for the models. They play a central role in the development of those models. Therefore, they are of paramount importance to the development of understanding in physics.

Checking the inferences based on an idea of P against the behavior of the actual P provides valuable feedback on how the idea of P relates to P's actual behavior. This feedback allows for a refinement of the initial model. These refined ideas, in turn provide a basis for further inference and testing, leading to further refinement of the model.

Therefore, tests that disconfirm ideas can be seen as more informative than test that confirm them, because the disconfirming tests indicate that the model needs to be revised. Therefore, these tests provide the basis for the refinement of the model. These ideas form the basis of Popper's philosophy of falsification (Popper, 1934), an approach to science that views the falsification of theories to be the main driver of progress in natural science.

By a constant cycle of inference, testing and revision, crude initial ideas about the hidden causes of P are gradually developed into a more refined model which better predicts the behavior of P. By having a more refined and more useful model of P, we have developed a better understanding of P. This is how understanding is developed in physics.

Physics uses mathematics to describe its theoretical understanding. Therefore, the theoretical models of physics are mathematical descriptions. Physicists use the language of mathematics to describe a structure that represents important aspects of P. This mathematical representation of P guides their reasoning about P.

The assumptions that underlie this idea of understanding

In the previous sections, I have shown that the field of physics seeks to understand our physical reality in a very specific manner. Physicists try to explain the phenomena of this physical universe by offering hypothetical hidden causes for those phenomena.

There is an important assumption that underlies this specific way of seeking understanding. This practice assumes that these hidden causes are the best way to understand our physical universe. This key assumption in turn, is based on assumptions about what the nature of this physical reality is like and what we can know of this reality. These assumptions are metaphysical. This means that they

cannot be proved or disproved by logic or physical evidence. Hence, they remain assumptions.

In the next sections, I will discuss the assumptions that underlie the idea of understanding in physics. First, I will discuss the assumptions that the field makes about the nature of physical reality. Next, I will explain how this specific idea of reality determines what we can know of this reality. Lastly, I will explain how these ideas imply that our physical world can best be understood in terms of such hidden causes.

Assumptions about the nature of reality

The practices of developing understanding in physics are based on the assumption that physical reality can best be understood through learning of its hidden causes. This view is built on a number of implicit assumptions about the nature of this physical world.

For instance, this view presupposes that things like 'hidden causes' exist in the first place, because phenomena somehow 'need' a cause to happen. It also presupposes that learning about the hidden cause of a phenomenon actually clarifies things, instead of making the rest of the universe *more* puzzling. In the next paragraph, I will discuss the most important implicit assumptions I believe to underlie this specific worldview.

Implicit assumption 1: All events must have a cause

In order to explain physical events in terms of hidden causes, physics has to make an important assumption about the nature of reality, which cannot be proved or disproved.

These hidden causes have never been observed by anyone. Still, a belief in them is taken to be justified. This is because the physical structure of visible matter cannot explain why a physical phenomenon P occurs. Therefore, the true cause of P is taken to lie elsewhere, beyond the visible.

But in itself, this conclusion does not follow from the premises. If X cannot be found here, it does not necessarily follow that X must be somewhere else. The one thing only follows from the other under the condition that X certainly exists. If it is not certain that X

exists, the fact that X is not found here could also mean that X is nowhere. Therefore, the justification for hidden causes is only valid when we are certain that a cause for P exists.

But we are not certain of this. We think that all events must have a cause, but this is just an assumption that we make. We have no factual basis for this. We do not know whether the nature of physical reality really requires all events to be caused by something else. Perhaps things simply happen, out of themselves, without any external reason. Perhaps, our physical universe just works the way it works, with no further explanation to it.

It is impossible for us to establish whether these hidden causes really exist or not. If they exist, they are invisible to us. We will never be able to observe whether there really is some hidden force behind the regularities observed in nature, or whether these regularities simply are the way they are. Therefore, the belief that all events must have a cause is a metaphysical belief: it cannot be proved or disproved by physical evidence. In order to justify seeking explanations in terms of hidden causes, physics has to assume that this belief is true.

Implicit assumption 2: The hidden causes of these events are universal in nature

If you want to explain physical phenomena with hidden causes in general terms, you must not only assume that these causes must exist, you must also assume that these causes are universal in nature. The influence they exert on matter must be uniform across time and space.

If these hidden causes do not possess a universal nature, the nature of causes would be specific to certain times, certain places or perhaps, even to singular events. Such specific causes can only explain their own specific effects. They cannot explain the behavior of matter and the physical phenomena that result from them in general terms. Learning about the hidden cause of one phenomenon would tell us little about the rest of the universe.

Therefore, a reality with non-universal causes would make the explanations of physics impossible. This means that physicists have to assume that the hidden causes of our world are indeed universal in nature.

Implicit assumption 3: These hidden causes explain, but they cannot be explained

The hidden forces of nature are taken to be the causal explanation of all the complex material interactions that happen in our physical world. These causes however, are not taken to have an explanation themselves.

These causes are a basic property of this physical world, meaning that they are just the way they are. Their nature cannot be further explained through the use of logic, or at least not in any trivial sense. It can also not be explained by invoking a possible cause behind the causes. If such a cause behind the cause exists, it is not taken to be accessible to human inquiry, not even in an indirect sense. From a scientific point of view, it would be useless to speculate about such a thing.

The statement that science cannot find a cause behind the causes may sound a bit counterintuitive. After all, it is perfectly possible that a future "theory of everything" shows that all fundamental forces of nature can be reduced to one and the same thing. This basic force of nature would then be the cause of all other forces. It appears then, that science *would* be able to find the cause behind the causes.

But actually, finding a single basic force is not the same as finding the cause behind the causes. The nature of such a basic force would explain the nature of the other forces. But the nature of basic force itself is not explained. Such a theory does not tell us where this single hidden cause comes from and why it is the way it is. Basically, this theory only reduces the number of mysterious causes from several to one. We would still be stuck with a mysterious, hidden force, for which we have no explanation.

Implicit assumption 4: The phenomena that result from these causes follow the rules of logic

These hidden causes are not only taken to be universal, they also are logical. The material behaviors and interactions they give rise to follow common rules of logic. If phenomenon P can only occur in the absence of phenomenon Q and there is a circumstance C under which hidden cause A will induce phenomenon Q, we can conclude from this that P will not happen under circumstance C.

Knowledge of this reality

The nature of our physical reality determines what can be known of this reality. As I will show, the specific worldview discussed above has some important consequences for the ways in which we can know this world. I will discuss the most important of these consequences.

Consequence 1: These causes are invisible to us

These causes lie beyond the visible. Therefore they cannot be detected by physical means, such as the human eye and scientific instruments.

Consequence 2: The nature of these causes cannot be deduced by us

These causes are taken to be a basic property of reality, which means that their nature does not follow from something else, like logic, or a cause behind the causes. They just are the way they are. This means that their basic nature cannot be deduced from something else.

Consequence 3: The nature of these causes can be inferred by us, from the behavior of physical matter

Because, according to assumption 4, these hidden forces cause stuff to behave in a manner that makes logical sense, we can use the behavior of that stuff to make logical inferences about the causes that underlie it.

Consequence 4: We can generalize the inferences we make to other phenomena

Because these hidden causes are taken to be universal, we can be sure that the inferences we make about them do not only hold for that one specific case that we are studying. They can be generalized to all other cases of the same phenomenon. Therefore we can use inferences from a limited set of cases to explain the behavior of phenomena in general terms.

From worldview to understanding

As I have shown, the field of physics makes a number of basic assumptions about the nature of physical reality. The basic assumptions in this worldview, in turn, imply that there is a certain way in which this reality must be understood. These ideas underlie the central idea of understanding in natural science. In physics it is believed that our physical world can best be understood through understanding its hidden causes.

In the next paragraphs, I will show how this idea of understanding follows from this specific worldview. Firstly, these assumptions imply that these hidden causes are the key explanation of this physical universe. Secondly, these assumptions imply that it is actually possible for us to obtain this key explanation. Therefore, these hidden causes form the most fertile avenue for understanding the physical world.

From the basic assumptions that physics makes about the nature of reality, it follows that 'key explanations' exist. These key explanations are explanations that are both complete and general. This means that they fully explain why events happen and do so in general terms. In this universe, these key explanations come in the guise of hidden causes. These hidden causes are key explanations because they are both complete and general explanations for the occurrence of physical events.

Firstly, these hidden causes are full explanations. As I have shown, visible matter alone does not suffice as a full explanation for physical events. However, according to implicit assumption 1, all things must have a cause. Therefore, a full causal explanation for these events must exist, even if it cannot be found in the visible. This explanation is taken to lie in a set of invisible causes. These

hidden causes explain events in a way that visible matter alone cannot. Therefore, these hidden causes are full explanations.

Secondly, these hidden causes are also general explanations. According to implicit assumption 2, these hidden causes are universal. The same causes are behind many different events that occur across time and space. This means that, the causes that explain one event simultaneously explain many other events also.

These hidden causes therefore, truly are key explanations. These hidden causes offer a complete explanation of why things happen and do so in general terms. Such explanations enable a far greater degree of understanding than descriptions of the physical could provide. Therefore, these causes are indispensable to understanding the physical world.

Not only do these assumptions imply that such a key exist, they also imply that it is possible to obtain this key. While, according to these assumptions it is not possible to have a direct understanding of those causes, an indirect understanding of them *can* be achieved.

From assumptions, 1 and 3 it follows that these causes cannot be observed (Consequence 1) and they cannot be deduced (Consequence 2). Therefore, they are inaccessible to us, making it impossible to have a direct understanding of those causes. But according to implicit assumption 4 these invisible forces cause matter to behave in logical ways. This makes it possible to learn about these causes in a more indirect manner: by using logical inference (Consequence 4).

Their nature can be inferred from visible phenomena and the way they behave, using a cycle of theoretical reasoning and empirical testing. Through this kind of inference, a more indirect form of understanding can be developed, embodying ideas about what could be, instead of what is. This allows us to have an understanding of the hidden causes.

The basic assumptions made by physics imply that the hidden causes are the key explanation, indispensable to understanding the physical world around us. They also imply that it is possible to obtain this key, through developing an indirect understanding of these hidden

causes. This is why physics believes that our physical world can best be understood through understanding its hidden causes.

Philosophy of science: discussing the search for hidden causes

As I have shown, the development of understanding in physics is tightly connected to a specific philosophical worldview. The field entertains a number of ideas about the nature of reality and the ways in which this reality can be understood. These issues have been discussed by philosophers also. I will conclude this chapter with a short overview of their viewpoints.

In his *Phaedo,* Plato already pointed out that the visible state of the physical is not sufficient to explain why a specific event occurs. If Socrates is sitting in an Athenian prison, the visible, physical state of his body helps us to explain why he is able to sit. But it does not explain why he is sitting, instead of standing or running. Like the case of the forest fire, a full explanation must be sought beyond the visible.

Plato believed that everything in the world had a specific purpose and was 'driven' to act out its purpose. Socrates was sitting because, at that moment, sitting was the action that best served him in acting out his goal or purpose. Maybe this comfortable posture allowed him to think more clearly and better prepare for what was coming (Plato, trans. 1966).

Aristotle believed that the purposes of different things could ultimately be reduced to just a few basic purposes. In a machine, every small screw has a specific purpose. This specific purpose can be explained in terms of the specific component it is a part of. And the purpose of this component, in turn, can be explained in terms of the purpose the machine is serving.

In a similar vein, the purposes of specific objects, animals, plants and people could be explained in terms of the purpose of the materials they consisted of. And the purposes of these different classes of matter, in turn, could be explained in terms of their purpose within the universe. Aristotle described these basis causes in his classical work *De Caelo,* among else (Aristotle, trans. 1922).

Aristotle thought that these basic purposes were necessary truths, which could be known by intuition. For him, this intuition provided a model for explaining the world. The occurrence of specific events could be deduced from our existing knowledge of the general. To understand an event, was to understand why it followed from these basic purposes. In the medieval Western world, his ideas became the dominant model for seeking understanding (Ladyman, 2002).

After the Middle Ages, people's ideas about seeking understanding slowly began to shift. In his *Novum Organum*, (1620) Francis Bacon criticized Aristotle's ideas. Bacon was not convinced that Aristotle's 'natural purposes' were necessary truths. There could be other explanations for the occurrence of physical things also.

This meant that the method of explaining things from pre-given purposes was deeply flawed. When you accept these purposes as a given, you will not consider the possibility that there might be other explanations for things, let alone look for them. You would never find out whether your initial ideas are justified or not, because you only develop explanations that confirm them. Therefore, Bacon argued, if you follow Aristotle's method, you will never be able to learn anything new.

If we want to learn about the world, we should stop explaining it in terms of our own pre-conceived ideas. We should see the world as it really is, not as we think it is. To observe the world in an objective manner, we should clear our minds from-pre-existing judgments. Only then are we able to observe what is *really* going on there.

Therefore, in order to develop true understanding, Bacon proposed that people should start with a careful and objective observation of the physical phenomenon. The scientist would base her ideas on her observations of the phenomenon only. Received wisdom, cultural norms, moral values or commitment to an existing idea should play no role here. The scientist had to be prepared to throw all her carefully developed ideas about a phenomenon out of the window as soon as the phenomenon seemed to contradict them.

By a careful and objective observation of a phenomenon, at different occasions and under different circumstances, people could develop an idea of the true mechanism behind it, which Bacon called 'forms'. According to Bacon, the invisible, true form of things produced the phenomena we observe in the world. By discovering the nature of these invisible forms, mankind would be able to explain and control the physical world.

These ideas provided the basis for our modern conception of natural science. As I have shown, the practice of inferring the visible from the invisible is still very present in modern physics. Over the years, the Baconian idea of doing science has received fierce criticism. This method is often presented as a method to 'let the facts speak', free of superstitious assumptions. However, as I have shown in this chapter, this method is based on quite some assumptions itself.

This method seeks to explain physical phenomena by claiming that they are caused by an invisible mechanism. Because this mechanism is invisible, its existence can never be proved or disproved by 'the facts'. Therefore the Baconian scientist has to assume that there is a mechanism behind everything we see in the world, despite the fact that no one has ever actually seen such a mechanism. That is quite an assumption, for a method that claims to be based on verifiable facts. David Hume (2003) among others, has criticized this form of science, for making unwarranted metaphysical assumptions.

Apart from issues with metaphysical assumptions, 'letting the facts speak' turns out to be harder than suspected in general. Bacon advocated a science based on objective observation. The actual behavior of the phenomenon should be guiding our judgment, instead of our pre-existing ideas. But according to several philosophers, such objective judgment may not at all be possible (Ladyman, 2002).

Hanson (1965) argues that our ideas of the world shape the way we perceive it. This means that are not able to perceive things 'as they are', without our view of the world getting in the way to some extent. Kuhn (1970) argued that scientific theories are never the product of pure, disinterested observation only. They are always

influenced by something else, such as the scientist's personal background, the beliefs of his community, historical circumstances and many other things. For Kuhn, science is anything but a disinterested, objective observation of the facts.

These criticisms, all pertain to science's ability (or inability) to provide us with objective truths. However, the philosophical school of pragmatism has pointed out that understanding does not need to be objectively true in order to be useful (Hookway, 2016). More recently, Boon endorsed a similar viewpoint (2009). Even if a theory does not accurately represent the true state of affairs, it still can yield accurate predictions. And if a theory enables us to accurately predict the behavior of phenomena, it can serve to explain and control these phenomena.

Bacon hoped that his new science would one day enable us to explain and control the world. The technological and scientific marvels of our modern world, suggest that, for a large part, we have succeeded in doing this.

References

Boon, M. (2009). Understanding in the Engineering Sciences: Interpretative Structures. In H. W. de Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding: Philosophical Perspectives*. Pittsburgh: The University of Pittsburg Press.

de Regt, H. W. (2009). Understanding and Scientific Explanation. In H. W. d. Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding, Philosophical Perspectives* Pittsburgh: University of Pittsburgh Press.

Hanson, N. R. (1965). *Patterns of discovery: An inquiry into the conceptual foundations of science*: CUP Archive.

Hookway, C. (2016). Pragmatism *The Stanford Encyclopedia of Philosophy.* Retrieved August 20, 2017, from https://plato.stanford.edu/archives/sum2016/entries/pragmatism/

Hume, D. (2003). *A Treatise of Human Nature* (J. P. Wright, R. Stecker, & G. Fuller Eds.). London: Everyman Paperbacks.

Kuhn, T. S. (1970). *The Structure of Scientific Revolutions* (2nd ed.). Chicago: The University of Chicago Press.

Ladyman, J. (2002). *Understanding philosophy of science*. Abingdon: Routledge.

Popper, K. (1934). *The Logic of Scientific Discovery*. London, United Kindom: Hutchinson.

Chapter 4: The development of understanding in mathematics

Introduction

In the previous chapter, I have discussed the understanding that is developed in physics. I have based my description on the work of Boon (2009) and de Regt (2009) about understanding in science. In this chapter, I will use their work to describe the understanding that is developed in mathematics.

First, I will explain what kinds of things mathematical theories describe. Then, I will show how these descriptions bring about understanding. I will show that they bring about understanding, because they help us to develop an understanding of our intuitive concept of structure. Then, I will explain how these mathematical theories help us to expand this intuitive concept. Next, I will show how my explanation of understanding of mathematics fits within the context of the larger debate in the philosophy of mathematics. I will argue why I have chosen this particular viewpoint to explain understanding in mathematics.

Mathematics as descriptions of structure

Theories in mathematics cover such a huge variety of subjects. While many people associate the field of mathematics with the study of numbers or spatial objects, mathematics does not confine itself to either of those topics. The field has developed theories about all kinds of things, ranging from geometry and algebra to communication, codes, conflict situations, waiting lines and even the theories of mathematics itself.

But while the subjects of mathematics seem hugely varied, they bring understanding in the same way. These theories do not bring understanding by describing actual things. They bring understanding by describing a particular type of structure. Mathematical theories define a basic structure and show what kinds of structural properties logically follow from this basic structure.

For instance, algebra describes the nature of mathematical equations by describing the structure that underlies these equations. The theory is a description of the relationships that exist between the

different operators and operands in an algebraic equation. Geometry describes the structure of the spatial relations that exist between spatial objects. Communication theory describes the structure of the different relations that exist between the basic properties of messages and communication channels. And category theory describes the structural pattern that underlies the theories in mathematics itself.

At this point, some readers may wonder why this would set descriptions in mathematics apart. After all, the concepts and relations of a theory in physics can be viewed as structures also. Doesn't that mean that theories in physics are also descriptions of structural patterns?

While I admit that these stories about structure sound very similar to each other, there is a subtle difference between them. The difference is that physics describes *other* things through providing structures, instead of describing structure itself. These structures are descriptions of actually existing, physical things. Structural descriptions in mathematics, on the other hand, do not describe the nature of actual things. They describe the nature of structure itself, independent of any actual instances of such a structure.

Some theories in mathematics may create the impression that they *do* describe the nature of actually existing things. This is because they are sometimes referred to as 'the mathematical theory of conflict' or 'the mathematical theory of communication'. These terms are a bit misleading, however. Like other theories in mathematics, these theories do not describe the nature of actual things. They describe the nature of a specific kind of structural pattern.

This doesn't mean that these theories have nothing to do with those actual things. The process of defining the basis of such a structural pattern is often informed by our ideas about these actual things. For instance, when we develop a mathematical theory of conflict, we define a type of structure that represents what we believe to be the main ingredients of actual conflicts: actors, available information, possible courses of action, expected pay-offs and the relationships that exist between those things.

But after we have defined our basic structure, our knowledge of the actual thing no longer plays any part in our theory. We derive the rest of the theory from this structural definition, showing what kinds of structural properties logically follow from such a basic definition. Therefore, this theory is a description of the properties of a certain type of structure, not a description of the actual thing.

How describing 'structure' brings about understanding

In the previous section, I have shown that theories in mathematics all describe a particular type structure. These descriptions enable us to expand our intuitive understanding of structure. We all possess an intuitive concept of structure. This basic concept of structure is indispensable for us to grasp the world. In the next paragraph, I will discuss this basic concept of structure and explain how this concept helps us to understand the world.

Our concept of structure

To us, it appears as if reality possesses a specific structure. This means that for us, it is possible to divide reality into several discernable objects, between which several specific relationships exist. Intuitively, we all have ideas of what this specific structure must be like. This means that the human mind somehow developed a conceptual understanding of this structure.

Understanding this concept requires an understanding of the notion of structure in general. Therefore, the basic component for this understanding is a concept of structure itself. This concept of structure, in turn, consists of an understanding of the notions of an *object* and a *relationship*.

Most of us seem to understand what an object is. We conceive of objects as separate entities, which are discernable from other entities. This means that we somehow understand the idea of existing as a separate entity, being discernable from other entities. Next to understanding what an object is, we also understand the notion of a relationship. This allows us to understand how discernable things can be connected to each other.

This understanding is not as trivial as it may seem at first. The concept of an object appears to be a very simple concept. But it is very hard to explain it without using terms that already presuppose an understanding of the notion of an object (like 'entity' 'discernable from' and 'other'). We cannot put into words what an object or a relationship is, but somehow, we have an understanding of these things.

Together, these two basic concepts enable us to grasp the concept of structure in general. This notion of structure allows us to interpret reality around us as an instance of a structure, consisting of several, discernable objects and relationships.

Next to understanding structure in general, we also developed an understanding of the specific structure we perceive in our universe. This means that we have developed an idea of the specific objects and relationships we discern in our reality.

For instance, we understand some of the objects in our universe as *material* objects. In our concept of structure, this entails a specific relationship with space and with time. Material objects occupy a certain region of space and time. My chair, for instance, takes up a certain amount of space and exists during a specific time period, from its creation until its destruction.

This means that these objects are related to other objects that exist in space and time. My chair exists somewhere in space. Therefore, it has a relative position to other objects that exist somewhere else in space: *below* my desk, *inside* my house, *east* from the prime meridian, for instance. Next to existing in space, my chair also exists in time. Therefore, it also has a relative position to all other objects existing in time. My chair was created *after* the middle ages ended, but is *older than* the Rotterdam central railway station and it has lasted *longer* than the statue built at Burning Man 2004.

In our concept of structure, not all objects occupy a region of space. Non-material objects like Harry Potter, Thursday or the number three cannot be found at a particular spot in this universe. Therefore, these objects are not spatially related to other objects.

Harry Potter cannot be above the number three, or adjacent to my chair.

Next to relationships of space and time, we also have an understanding of other sorts of relationships. For instance, we understand the relation of similarity. We can relate objects to each other by being *similar* or *dissimilar* in respect to some property. We understand the concept of magnitude, which means that we can relate objects by being *greater than* or *smaller* than each other. And we understand the relationship between the *part* and the *whole*. This means that we can view objects as a whole, consisting of smaller parts, which can be seen as objects themselves.

We understand that this part-whole relationship works in two directions. Therefore, objects can be viewed as part of a larger object also. And we understand that we can extend this part-whole relationship indefinitely: My house is part of my street, my street is part of my neighborhood, my neighborhood is part of my town, my town is part of my region and so on.

Our understanding of structure is an understanding of relationships

My brief description of our concept of structure may leave the philosophically inclined reader with many, many questions. Why do material objects exist in time and space? Can some non-material objects also have an existence in space? Can we really lump abstract concepts, made-up fictional characters and non-tangible parts of the material universe in the same category? Do objects like abstract concepts and fictional characters even exist independently of us? And if so, how do those things 'exist'?

However, for this concept, these questions are not very relevant. These questions concern an understanding of the *nature* of reality. But this concept is not an understanding of the actual nature of reality. It is an understanding of how *we* can relate things to each other.

For instance, this concept does not involve an understanding of the actual nature of space. Instead, it provides an understanding of how objects are related to each other in space. We understand how things can be above, inside or adjacent to each other in space. But space,

as a phenomenon in itself, is not understood through this concept. In a similar vein, this concept does not tell us what 'non-material objects' are and how they can be said to exist. It only enables us to understand how we can relate these objects to other objects.

Where does this notion come from?

At first sight, our understanding of structure may appear to be a trivial understanding. But although basic, this concept is not trivial at all. Like the concept of objects and relations, concepts like *similar*, *before*, *adjacent to*, *part and whole*, or *greater than* cannot be explained in terms that do not already presuppose an understanding of these concepts. And yet, an understanding of these concepts comes natural to us.

Even non-human animals possess an understanding of these concepts, to some degree. For instance, Hunt, Low, and Burns (2008) showed that wild robins can discern between small quantities of tasty mealworms, like three and four worms. The robins preferred holes with larger quantities of worms. And when the researchers secretly removed some of the worms from the holes, the robins kept looking for the 'missing' worms. This means that these robins understand the notion of an object and a collection. They also understand that collections can be larger or smaller than other collections of objects and they can remember how big particular collections are. And, as young as zero to three days after birth, human infants already grasp that the concept of magnitude applies to space, time and quantity (de Hevia, Izard, Coubart, Spelke, & Streri, 2014).

Non-human animals have no language to formulate and communicate ideas in, which means that they cannot teach abstract concepts to their young. Human infants also do not possess language. And at three days after birth, they almost have little to none experience of the world. But still, they are able to grasp abstract concepts like quantity and the relation between quantity, time and space. Therefore, it seems unlikely that they have learned these concepts from experience.

In 1787, Immanuel Kant already argued that this concept is not learned from experience. Kant views our concept of structure as a

precondition for learning from experience. Without this concept, we would be unable to make any sense of the world.

This makes it impossible to learn this concept from experience. It would require already possessing the concept that we have yet to learn. Therefore, our concept of structure must come from somewhere else. It must be an innate part of the human cognitive system. This means that we do not have a concept of structure because we perceive it in the world, but we perceive the world because we have a concept of structure.

According to Kant, reality itself does not come to us as a structure. Our senses do not really perceive objects and relationships. They only perceive sounds, vibrations, smells, colors, shapes, bodily sensations and accelerations. It is our brain that is actively creating a world full of structure out of these inputs. It views these sensory inputs as if they were coming from a world full of discernable objects.

Changes in sensory stimuli are interpreted as signifying differences in objects. For instance, to our brains, changing color stimuli signal the presence of a discernable object, or a discernable part of an object. And if the intensity of background noise suddenly changes, our mind interprets this change as coming from a distinct source: an object that is producing a noise (a dog barking) or blocking it (someone closing the door).

This process of interpretation cuts the stream of stimuli up into several discernable objects. Then, our mind uses its understanding of relationships to draw connections between these objects. It may connect these objects through time, space, similarity, magnitude or being part of a greater whole. By drawing these connections, our brain changes the original stream of sensory inputs into a world of discernable, related objects. This process lets us perceive reality as full of structure.

Such a perception of reality is necessary for us, in order to make any sense of the world. This is because this lets us perceive reality as consisting of *objects* and *relationships*. These objects and relationships are the key ingredients for developing an

understanding of the world, because they enable us to reason about reality. In order to make any meaningful inference, we always need an object to reason *about* and a relationship to reason *with*. Therefore, objects and relationships are indispensable to drawing inferences about our perceptions of reality. Without such objects and relationships, we would not be able to draw any meaningful conclusion from our perceptions. We would not be able to learn anything from our sensory experiences and we would never develop an understanding of reality.

In order to be able to make sense of the world, our mind needs to understand the world in terms of a structure, consisting of a set of discernable objects and relations. This is also a core tenet of Mieke Boon's work about scientific understanding. In order to gain an understanding of a thing, we need to have an interpretative structure of it (Boon, 2009). Therefore, the notion of structure is fundamental to our ability to understand.

If structure is our brain's way of understanding the world, it comes as no surprise that we perceive structure everywhere. All matters that we can think of possess structure: physical objects, hypothetical objects, abstract objects and even objects that are totally fictional. For our minds, there is no other way to conceive of these objects. If things also exist in different manners, we are not able to comprehend this. The basic idea of structure underlies all of our thinking.

How mathematical theories expand this intuitive concept of structure

In the previous sections, I have shown that theories in mathematics are descriptions of structure. These descriptions bring about understanding, because they enable us to expand our intuitive concept of structure, which we use to make sense of the world.

In this section, I will show how mathematical theories enable us to extend this intuitive concept of structure. Although our concept of structure is innate, it is possible to extend it.

Our intuitions form the basic core of our understanding of structure. They provide us with an understanding of how objects can

be related to each other. But we do not have an intuitive understanding of all the things that follow from these basic ideas. This requires reasoning about these ideas. We can use our intuitive understanding of structure as a starting point for further logical inference.

Theories in mathematics enable us to do this. The structures they describe are explicit formulations of our intuitive ideas of structure. They describe our fuzzy intuitions as clear structures, consisting of a clear set of concepts and relations. With these concepts and relations, we can reason more precisely about these intuitive ideas, increasing our understanding of them.

Mathematical theories as interpretative structures

In the previous chapters, I discussed Boon's account of scientific understanding. I explained that in order to understand a thing we need to have an interpretative structure of that thing. Such an interpretive structure provides a clear set of concepts and relations which enable us to reason about that thing. Therefore, we can develop an understanding of it. I showed how theories in physics provide interpretative structures of physical phenomena, which enable us to understand those phenomena.

When it comes to understanding in mathematics, it might not be entirely clear why we need such an interpretative structure. After all, as I have shown, the field of mathematics is all about further developing our intuitive understanding of the structure of our perceived reality. So, apparently, this is an understanding we already have.

And having an understanding of this structure, in turn, implies that we also understand which objects and relationships can be discerned in this reality. In the previous sections, I explained that this is indeed the case. We intuitively understand how objects are related to each other through time, space, quantity and other things.

So why would we need an interpretative structure, to further develop this understanding? Why is it necessary to borrow the concepts and relations from another theoretical structure? If we want to reason about our intuitive concept of structure, wouldn't it be much

simpler to use the objects and relationships from this concept of structure itself?

But this is not as straightforward as it seems. If we want to reason about these objects and relationships, we need to know what we are talking about. Therefore we need a clear understanding of *what* objects there are and *how* those are related. And our intuitive concept of structure does not provide us with a *clear* notion of structure.

Our basic understanding of structure is an implicit, intuitive form of understanding. It enables us to intuitively 'see' how things are related, when we encounter them in our world. But it is an understanding without words. Therefore, it does not provide us with a body of formal knowledge that specifies which types of objects there are and how they are related. And if we cannot specify the relationships between objects, we lack a clear starting point for making further inferences.

If we want to use these intuitions for further inferences we have to transform them. Our initial, intuitive understanding of how things in the world relate to each other must be changed into a clearly described set of objects and relations, in order to have a starting point for further inference. This is what mathematical theories do. They serve as an interpretative structure for developing a further understanding of these intuitive ideas.

Mathematics, understanding and philosophy

In this chapter, I explained how the field of mathematics can provide new understanding. In the philosophy of mathematics, this question is the subject of much heated debate. Therefore, it appears as if it is very difficult to provide a satisfying answer to this question. However, I believe that this is not as complicated as it may look at first.

In the next section, I present an overview of this debate. Then, I will argue that much of this philosophical debate about is based on a misunderstanding. This misunderstanding is caused by confusing mathematics with natural science. Then, I will explain why I have chosen to explain mathematical understanding as an understanding of

structure. Because this account does not suffer from this confusion, it is able to explain the relationship between understanding and mathematics in a satisfying manner.

The philosophical debate about mathematics

Over the years, philosophers and mathematicians have sought to explain how mathematics brings about understanding. They tried to find out what sorts of things in this world are explained by mathematical theories. They also wanted to establish what mathematical theories say about these things.

This debate was not solely motivated by a philosophical interest in mathematics. People also sought a justification for the theories of mathematics. Natural scientists justify their claims with evidence and logical reasoning. But what reason do we have to accept the statements in mathematical theories as true? In order to explain what makes a statement true, one needs to know what this statement is about. Therefore, people needed mathematics to have a clear subject. Several competing accounts of the subject of mathematics have been put forward (Horsten, 2016).

Mathematical realists argue that the science of mathematics is about real aspects of objective reality. There are different ideas about what aspects of this reality are being studied. *Mathematical platonists* argue that mathematics is about mathematical objects, that exist as abstract things outside of time and space (Linnebo, 2017). However, it is difficult to understand how these objects could exist outside time and space. And if these objects exist out of time and space, it is impossible for them to have a causal link with our temporal, physical universe. Therefore, it is hard to see how we, as physical beings in a physical universe, can have knowledge of these 'mathematical objects'(Horsten, 2016).

Another version of mathematical realism is *mathematical naturalism*, as put forward by Quine (1951). In this view, there is no principal difference between mathematical facts and empirical facts, like the facts discovered in physics. But if that is the case, it is hard to explain why our mathematical intuitions seem so very different from other facts discovered trough experience. To us, statements like

'between any two points, a straight line can be drawn' feel as necessary truths, not as something we learned over time, after encountering many points with straight lines between them.

Others have argued that mathematical theories are not about 'the mathematical' but describe something else. For instance, in the late 19th century, the school of logicism tried to prove that mathematical theories actually describe a sophisticated type of logic (Horsten, 2016). However, their attempts to reduce the foundations of mathematical theories to more basic logic have failed.

In the early 20th century, the *formalist school* sought a solution to this problem in another direction. For the formalist, mathematics is not 'about' anything. Therefore, mathematical theories aren't about anything either. These theories simply describe a certain structure and do not refer to anything outside this structure. They can be understood as meaningless, made-up structures (Zach, 2016).

These made-up structures acquire meaning by being 'borrowed' to describe other things. For instance, the system of Euclid's geometry was in itself meaningless. It only acquired meaning because it could be used to describe actual geometrical spaces. This is what makes the science of mathematics useful, despite not being about a meaningful subject itself.

The formalist account of the subject of mathematics is not without problems either, however. Mathematicians prefer games that correspond to their own mathematical intuitions. Apparently, they want mathematical theories to represent their mathematical intuitions in a correct fashion. This would mean that these theories *do* refer to something 'mathematical' outside themselves (Horsten, 2016). This means that these mathematical structures aren't as meaningless as the formalists would like them to be[2].

There is a lot of confusion and debate, about how mathematical theories bring about understanding. Because of this, it seems hard to provide a satisfactory answer to this question. However, I think that much of this confusion stems from a misunderstanding of

[2] I will discuss the formalist school and their approach to justification in more detail in chapter seven.

mathematics as a science. Therefore, I believe that this question is easier to answer than we think.

The confusion between mathematics and natural science

The root of this misunderstanding lies in a tendency to confuse the development of understanding in mathematics with the development of understanding in physics. In many ways, the natural sciences, especially physics, are being seen as 'model sciences'. There is a good reason for this. As scientific practices, the natural sciences are easy for us to understand.

The methods and practices in those fields are tightly connected to a specific view of reality. This specific of reality places the practices of these fields within a clear context. The development of understanding in these fields can be understood as the development of understanding of the Baconian universe. In the Baconian universe, the visible is caused by the hidden, so an understanding of the visible is developed through studying the hidden.

These 'Baconian' sciences provide a clear and exemplary model for the development of scientific understanding. Therefore, I believe that it is implicitly assumed that the development of understanding in other fields must be understood in the same manner. Like the Baconian field of physics, other scientific fields are also taken to bring about understanding by studying a special class of things, existing in this reality. These things are not necessarily tangible, visible things. They can exist in other ways too. But they *are* a part of our objective reality.

But the field of mathematics doesn't have a clear picture of what the world is like. Therefore, the field cannot tell us what things in this world it studies to create new understanding. Despite the lack of such a story, the field is very able to provide new understanding. This is the point where people start to get confused. They think that mathematics must be explaining *something* that exists in this world. Otherwise it would not bring about understanding. So they start looking for that thing.

Mathematical realists try to explain mathematical understanding by claiming that the objective universe is endowed with a mathematical

aspect. Mathematical theories would bring about understanding because they describe and explain this aspect. These people face the difficulty of having to explain where this mathematical aspect fits in with the rest of our objective reality. Others, like the *logicists* and the formalists, try to get around this by taking a more modest approach. They try to connect the theories of mathematics to the things we already know to exist and are familiar with. They run into trouble too, however, because these things never quite fit the theories of mathematics.

Explaining how mathematical theories enable us to develop new understanding is very difficult, because many people seek to explain this in the wrong way. They think that mathematical theories bring about understanding because they explain some object or aspect of our reality, like Baconian sciences do. But mathematical theories do not help us understand by describing reality outside us. As I already explained, they are not about the nature of reality. They explain how *we* can relate things to each other.

These theories further expand our intuitive concept of structure. This intuitive concept of structure, in turn, helps us to make sense of reality. As I have demonstrated in this chapter, the development of understanding in mathematics can be explained as the development of this intuitive concept. That is why I have chosen this approach for my discussion of understanding in mathematics.

My viewpoint is part of a larger tradition of explaining mathematical understanding. Kant (1998) already argued that mathematical intuitions are a product of the human cognitive system. Before the Second World War, the school of *mathematical intuitionism* argued that Kant's ideas could be helpful in explaining mathematical understanding. (Iemhoff, 2008). And more recently, Lakoff and Núñez (2000) proposed a cognitive science of mathematics.

References

Boon, M. (2009). Understanding in the Engineering Sciences: Interpretative Structures. In H. W. de Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding: Philosophical Perspectives*. Pittsburgh: The University of Pittsburg Press.

de Hevia, M. D., Izard, V., Coubart, A., Spelke, E. S., & Streri, A. (2014). Representations of space, time, and number in neonates. *Proceedings of the National Academy of Sciences, 111*(13), 4809-4813.

de Regt, H. W. (2009). Understanding and Scientific Explanation. In H. W. d. Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding, Philosophical Perspectives* Pittsburgh: University of Pittsburgh Press.

Horsten, L. (2016, Winter 2016). Philosophy of Mathematics. *The Stanford Encyclopedia of Philosophy.* Retrieved August 19, 2017, from https://plato.stanford.edu/archives/win2016/entries/philosophy-mathematics/

Hunt, S., Low, J., & Burns, K. (2008). Adaptive numerical competency in a food-hoarding songbird. *Proceedings of the Royal Society of London B: Biological Sciences, 275*(1649), 2373-2379.

Iemhoff, R. (2008, Winter 2016). Intuitionism in the Philosophy of Mathematics. *The Stanford Encyclopedia of Philosophy.* Retrieved August 20, 2017, from https://plato.stanford.edu/archives/win2016/entries/intuitionism/

Kant, I. (1998). *Critique of pure reason* (P. Guyer & W. W. Allen, Trans. P. Guyer & W. W. Allen Eds.). Cambridge Cambridge University Press.

Lakoff, G., & Núñez, R. (2000). *Where Mathematics Comes from: How the Embodied Mind Brings Mathematics Into Being* New York: Basic Books.

Linnebo, Ø. (2017). Platonism in the Philosophy of Mathematics. *The Stanford Encyclopedia of Philosophy.* Summer 2017. Retrieved August 20, 2017, from https://plato.stanford.edu/archives/sum2017/entries/platonism-mathematics/

Quine, W. V. O. (1951). Two Dogmas of Empiricism. *The Philosophical Revieuw*, 20-43.

Zach, R. (2016, Spring 2016). Hilbert's program. *The Stanford Encyclopedia of Philosophy.* Retrieved August 19, 2017, from

https://plato.stanford.edu/archives/spr2016/entries/hilbert-program/

Chapter 5: Describing understanding in computer science

Introduction

In the next chapter, I will describe the specific kind of understanding that is developed in computer science. In this chapter, I will explain the methodology of this description. I will discuss the method I have chosen to describe this form of understanding and I will explain why I have chosen this method.

Describing understanding in computer science

According to Kuhn (1970) each scientific field is based on a specific underlying paradigm. This paradigm shapes the way a field seeks understanding. Therefore, a field's way of understanding can be explained by the paradigm underlying it. This means that, if I want to describe the understanding developed in computer science, I need to study its underlying paradigm.

Such a paradigm consists of a set of ideas. These are ideas about what kind of thing the field is studying and how this thing should be understood. The ideas of this paradigm provide the field with a clear set of research goals, a common language to formulate problems in and a number of solution strategies. Therefore, they shape the field's specific way of seeking understanding.

In his work, Kuhn mainly discusses fields in empirical science. But non-empirical fields have underlying paradigms too. For instance, I have shown how mathematics develops its own specific way of seeking understanding. The field of mathematics shares certain ideas about understanding structure. In the field of mathematics, structure is perceived to be an abstract concept. This abstract concept is to be understood through the creation of formal descriptions. These ideas, about understanding structure, form the paradigm for the field of mathematics.

Describing current paradigms

The beliefs in such a paradigm show us how a field seeks to understand its subject. But studying these paradigms may not always be straightforward. Although the practices of a field are shaped by

its underlying paradigm, these practices may not be the best place to look when trying to understand this paradigm.

This is because the beliefs in a paradigm are often taken for granted. They are considered to be a point of departure for study and discussion, not the subject of study and discussion. Therefore, they are almost never made explicit in the field's actual research practices.

This is similar to questioning whether or not the sun will rise again tomorrow. People want to get ahead with making plans for tomorrow, so they simply assume that the answer to this question is yes. This prospective answer is a starting point for further discussion, not a part of the discussion itself. From a philosophical viewpoint, it is a perfectly justified question to ask. But philosophically questioning all your core beliefs won't get you anywhere anytime soon when you want to get ahead with things.

In a similar vein, scientists are not interested in questioning the foundations of the field all the time, because that doesn't get them ahead in the research that they are doing. Therefore, these foundations are not part of the scientific discussion.

This means that the field's textbooks and research papers will rarely, if ever, explicitly discuss those beliefs themselves. These textbooks and research papers will only discuss how to solve problems *according to* those beliefs. Therefore, a field's current research practices may not provide much insight into the specific sort of understanding it develops.

These fundamental beliefs only become part of the discussion at the formation of a new paradigm, when it is still unclear how something should be understood. Scientific fields can be in need of a new paradigm because the old paradigm starts to break down, or because they never had one in the first place.

Because these core beliefs are only part of the discussion during the formation of a paradigm, the formation period is more informative than the paradigm itself. Therefore, if I want to discuss the current paradigm in computer science, I need to study

the period in which this paradigm developed. Therefore, my description of understanding in computer science will consist of a historical study, which shows how computer science's current way of understanding emerged.

Studying the history of a paradigm

In an interesting historical study, Nofre, Priestley, and Alberts (2014) describe the emergence of the first paradigm for the field of computer science.

These authors show that, in the second half of the 1950's, people's understanding of computing underwent an important change. They argue that this new way of understanding of computing provided the first paradigm for the scientific study of computers. Around this new paradigm, the field of computer science emerged. Therefore, this new way of thinking allowed the field of computer science to develop into a separate, scientific discipline.

These authors describe the emergence of computer science as the emergence of a new paradigm. This makes their study very relevant to my thesis. Therefore, this study will be a foundation for my historical research.

My use of sources

In my history study, I have made use of two kinds of sources. Primary sources are materials that were published at that time, like books, articles and proceedings, intended to share the state-of-the art of programming knowledge at that time. Secondary sources are materials that were published at a later moment, meant to reflect on that time, like histories of programming and personal memoirs of computer pioneers.

When it comes to drawing a picture of a particular time period, primary and secondary sources each have their own advantages and disadvantages. One advantage of primary sources is that they are a more faithful reflection of people's understanding of a problem at a particular time.

Often, people have the tendency to fill in their recollection of the past with hindsight knowledge. Many histories of scientific

discoveries are portrayed as a search for an answer to a well-defined and well-understood problem, necessarily ending in finding 'the' solution. But at that time, for the people studying the problem, it might not have been so obvious what the problem was. Perhaps, people did not even realize there was a problem in the first place. And if the problem is not so obvious, that which was later hailed as 'the' solution, might at first, not have been recognized as a solution at all. Only in hindsight, a clear picture of the problem and its solution will emerge.

Primary sources, originating from the time period one aims to inscribe, do not have this distortion. These sources are a reflection of how people at that time, understood their field. These sources can therefore be very revealing. For instance, terms used in these sources tell a lot about the concepts underlying them. Because these concepts are often not explicitly explained, one has to find them by reading between the lines, looking how and where different terms were used. In this way, one can trace back how people at that time understood the things they were working on.

While primary sources are informative, they do not provide a complete picture. From the early days of programing, there is not very much literature available. Although many computers were used at research institutes and universities, computers themselves were not yet seen as an object of scientific study, so developments in computing were often shared informally, instead of publishing scientific books and papers about it.

The literature that is available is often meant to communicate a particular new idea. These sources are not meant to give a broad overview of the developments at that time. Therefore secondary sources are needed also. In this chapter, I use several secondary sources Nofre et al. (2014) Campbell-Kelly, Aspray, Esmenger, and Yost (2014); Nijholt and van den Ende (1994) have done some important historical research, placing the development of programming languages in a wider perspective, involving the interest of different actors, like businesses and computer experts who wanted to claim legitimacy for their field as a separate science. The research by Nofre et al. (2014) is important because it describes

the conceptual change that led to the emergence of a new paradigm in computer science.

A specific conception of computer science

In their article, Nofre, Priestley and Alberts discuss the emergence of computer science. However, their history is not *the* history of computer science. It is the history of one particular conception of computer science. As I will show, Nofre, Priestley and Alberts attach a very specific meaning to the term 'computer science'. These authors define computer science as the *formal study of computing*.

Not everybody agrees with this specific vision of computer science. Some people may think that this view of computer science is too narrow. This discipline encompasses more than formal, theoretical research. Many other accounts of the history of computer science define this field in a much broader sense.

However, as I will explain, I believe that there are good reasons to prefer the account of Nofre et al. (2014) over alternative accounts of the history of computer science.

Nofre, Priestley and Alberts: Computer science as the formal study of computing

Nofre, Priestley and Alberts view computer science as the formal study of computing. This particular view is revealed by the starting point they choose for their history.

These authors could have chosen to define the introduction of the computer as the starting point for computer science. After all, that was the period when people started designing them, building them, working with them, learning about them. The first modern, stored-program computers were introduced at research institutions shortly after World War II. And the more primitive ancestors of modern computers, programmable calculation devices and automatons, had existed since antiquity[3].

Another starting point could be the moment when the theoretical principles behind modern, automated computing were first formulated.

[3] For instance, the Antikythera mechanism is a mechanical device that could be used to predict astronomical positions. This mechanism was found in a shipwreck near the Greek island of Antikythera and is estimated to be more than 2000 years old.

This happened before the first modern computers were built, in the 1930's (Church, 1936; Turing, 1936). These theoretical principles are built on mathematical foundations, from the early 20th and 19th century, which, in turn can be traced back to older work. These are all important milestones, all of which can be viewed as being the origins of the developments in modern computer science.

But that is not what Nofre, Priestley and Alberts do. These authors make a different choice. They claim that computer science did not emerge until the second half of the 1950's, after people's thinking about computers underwent a radical shift. People realized that computing could be understood as an abstract concept, which could be understood in formal, mathematical terms.

This means that these authors do not think that computer science emerged with the introduction of the computer. They also do not think that the field emerged with the formulation of the first principles of automated computation. The authors think that computer science emerged with a new way of thinking about computers. Therefore, for these authors, computer science is not defined by having knowledge of computers. It is defined by a specific *way of thinking*. And this new way of thinking is the formal way of thinking.

Why the history of the formal study of computing is relevant to my thesis

These authors have some compelling reasons to equal the notion of computer science with this particular way of thinking. This new way of thinking changed the practical field of computing into a science. All theoretical branches of computer science are based on this paradigm of formal thinking.

But not everybody may agree with their vision of computer science. People may find their conception of computer science too narrow. They could point out that computer science encompasses much more than formal, theoretical research. Therefore, when you equal computer science with the formal study of computers, you reduce an entire field to just one single facet.

However, as I will show, the narrow facet that these authors discuss is exactly the aspect that I am interested in. In my thesis, I want to show how computer science can develop scientific understanding. This means that I need to understand how computer science works as a science. Therefore, in this chapter, I want to describe the emergence of *computer science as a science*.

In their study, Nofre et al. (2014) show that the emergence of computer science as a science coincides with the emergence of the formal paradigm. Before the emergence of this new form of thinking, people also developed valuable insights about computing. Many of these insights became the foundations for the field of computer science. In their study, Nofre et al. (2014) show how these insights led to the emergence of this new way of thinking.

But only when the formal way of thinking became commonplace, the practical discipline of computing could develop into a field of scientific study. This new way of thinking provided a shared understanding of computing. This, in turn, enabled the development of a set of shared goals and research methods. In other words: the field of computing acquired its first scientific paradigm. From this moment on, people in computing started doing science.

By describing the emergence of computer science as the formal study of computers, these authors describe the emergence of computer science as a science. This is only a small piece of the puzzle, but it is the piece that I need. This makes their study very relevant to my thesis.

To what extent does a story from the past reflect computer science of today?

I have chosen to describe the understanding developed in computer science with a history study describing the emergence of computer science in the 1950's. As explained in the previous sections, there are compelling reasons to do this. But this approach may also raise some critical questions. To what extent, is the understanding that Nofre et al. (2014) describe, representative for the understanding developed in computer science today?

In their article, Nofre et al. (2014) argue that the conceptual change they describe, set the scene for computer science as we understand it today. Also, Priestley (2011) has argued that these changes introduced a new way of thinking about programming, which still influences our ideas about programming today. The practices of modern computer science seem to support these claims.

Modern-day theoretical computer science still studies computation as an abstract, formal concept. Active subjects in the field are, for instance programming language theory, the complexity of computational problems and the use of formal, mathematical methods to verify the correctness of computer programs. According to Priestley (2011), these were the problems that were defined as important at the beginning of the new field of computer science.

The understanding of computing described by Nofre et al. (2014) is not only dominant in the theoretical fields of computer science. The more applied branches of computer science focus more directly on problems related to the effective functioning of real-world systems. But they also study aspects of computation as an abstract, mathematical concept.

For instance, the field of databases is concerned with the efficient storage and retrieval of data, which is an essential part of digital computation. They study these databases as logical structures and seek to describe their practices with several mathematical languages. The subfield of distributed computing focuses on how mathematical computation procedures can be broken into different parts and be distributed between different processors. And the field of computer graphics studies mathematical procedures which effectively process and display graphical images.

Therefore , I have very good reasons to believe that the paradigm described by Nofre et al. (2014) does not only reflect the development of understanding of computer science in the early 1960’s. It also reflects the development of understanding in computer science today.

References

Campbell-Kelly, M., Aspray, W., Esmenger, N., & Yost, J. R. (2014). *Computer: a History of the Information Machine*. Boulder, Colorado: Westvieuw Press.

Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics, 58*(2), 345-363.

Kuhn, T. S. (1970). *The Structure of Scientific Revolutions* (2nd ed.). Chicago: The University of Chicago Press.

Nijholt, A., & van den Ende, J. (1994). De Geschiedenis van de Rekenkunst: Van Kerfstok tot Computer (The History of Computing: From Tally-Stick to Computer).

Nofre, D., Priestley, M., & Alberts, G. (2014). When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960. *Technology and culture, 55*(1), 40-75.

Priestley, M. (2011). *A science of operations: machines, logic and the invention of programming*: Springer Science & Business Media.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math, 58*, 345-363.

Chapter 6: The development of understanding in computer science

Introduction

In this chapter, I plan to describe how computer science develops new understanding. In the last chapter, I have shown the best way to describe understanding in computer science is to describe how this way of understanding emerged. This means that my chapter about understanding in computer science will be a historical study. In this study, I will describe the development of the paradigm of modern computer science.

For a large part, my history study will be based on the work of Nofre, Priestley, and Alberts (2014). They show how practical issues with writing computer programs set in motion a chain of events. Together with economic developments and political ideals, these events eventually changed our understanding of computers and programming languages. This new way of thinking became a new paradigm for the new field of computer science.

First, I will use the article of Nofre et al. (2014) to discuss how people's understanding of programming languages changed during the second half of the 1950's. Then, I will show how a combination of technical inventions in computer programming, economic developments and political ideas brought about this change. Next, I will argue how exactly, these developments changed our understanding of computing. Then, I will argue how this new way of thinking led to the emergence of the new field of computer science.

Programming and Language, a Conceptual Shift

Since the first electronic, digital computers, people have been talking about programming these machines in terms of 'language'. Early computer experts explained the activity of programming as 'explaining a problem in the language the machine can understand'(Nofre et al., 2014). The concepts of programming and language still seem intimately related today. Programming is done in a *programming language*.

However, the use of the word 'language' hides an important conceptual shift. The pioneer's 'language' was not the same as the

modern programmer's use of the same word. In their article Nofre et al. (2014) show that pioneers and modern-day computer experts have very different ideas about the activity of programming. The concept of programming of the early pioneers was closely connected to their view of the computer itself, whereas our modern view of computing is more abstract. I will discuss how these pioneers viewed programming and contrast this with the present-day view of this activity.

The Computer: A Clever Robot Servant

When the first electronic, digital computers came into use, these machines were portrayed as almost sentient beings. These machines were lightning-fast calculation wizards, that could crack difficult mathematical problems many times faster than even the smartest humans could.

The most advanced of these machines were the *stored-program computers*. Earlier computers had to be manually set up to perform specific calculations, by plugging several switches. The stored-program computer could be instructed in a much simpler way. It was able to 'read' a program of coded instructions, provided along with its input data. Depending on the data it was fed, it could even modify these instructions. This provided these stored-program computers with a flexibility never encountered before in a machine. The concept of the stored-program computer would become a blueprint for the modern computer.

Therefore, it was no surprise that people were impressed by these machines. Newspapers wrote about 'electronic brains' and 'robot calculators' (Berkeley, 1949; Nijholt & van den Ende, 1994; Nofre et al., 2014). Not only the popular press, but also the builders and inventors of the computers used human-like analogies when talking about these machines. For instance: John von Neumann modeled his design of an electronic computer after human neurons, (Nofre et al., 2014) Grace Hopper expressed the efforts towards simplifying programming computers in terms of the 'education' of a computer (Hopper, 1952) and Alan Turing asked himself in a famous paper whether machines could 'think' (Turing, 1950).

This conception of the computer has its roots in the fascination the American people had with robots since the Great depression (Nofre et al., 2014). Robots were seen as the hallmark of technological progress. Also, viewing the computer as a machine with human-like capabilities expressed the hope that the electronic computer could someday replace the human computers. At that time, large computational tasks were carried out by large groups of female office workers (Nofre et al., 2014)

This particular conception of the computer influenced how people thought of operating computers. Ordinary machines are operated, but the computer, like a clever human servant, received *instructions* about the task to be carried out. The use of the word instruction is quite common in the early literature about computers. It can be found in several papers and books: (Berkeley, 1949) (Burks, Goldstine, & von Neumann, 1946; Campbell, 1952; Carr III, 1952; Hopper, 1952; Hopper & Mauchly, 1953; Laning Jr & Zierler, 1954; Levin, 1952) to name but a few.

The Language the Machine Can Understand

According to Nofre et al. (2014) it is this anthropomorphic view of the computer that brought forth the first use of the word '*language*' when talking about computers and programming.

Of course, beneath all its cleverness, the computer was an ordinary electrical device (albeit very advanced for that time). It worked by manipulating patterns of electrical signals. These were manipulated according to a set of rules, which were hard-wired in the machine. In themselves, these patterns meant nothing. But these pattern manipulations were isomorphic to calculations of mathematical functions. Therefore, the output of the computer could be interpreted as the output of the represented function. This left the burden of doing the actual calculation to the computer. The current patterns in a computer consisted of a series of high and low voltages, which were used to represent zeroes and ones. These zeroes and ones, in turn could be used to represent all kinds of other information.

If one would get this machine to perform a series of calculations, (run a program) one had to feed it a series of these current patterns. The computer would then read this series as a set of input data and a set of instructions. It would, according to its inner hard-wired rules, carry out the corresponding operations on the input data, and deliver the desired output.

People extended the clever robot servant analogy to this job (preparing the computer for specific programs). Just like a human servant who only spoke a foreign tongue, the robot servant also had to be addressed in its own language. People started to refer to this as the *machine language*. The term machine language was in use as early as 1947 (Nofre et al., 2014).

The term 'machine language' does not pertain to one language, but rather to a set of internal languages. Computers have different processors, with different sets of hard-wired instructions available, and different memory sizes and arrangements. To stay with the robot servant analogy: different servants spoke different languages. The term 'machine language' is a term for the collection of the different internal languages of different machines.

Programming as Translation

These ideas about the computer (as a clever robot servant which had to be instructed in its own machine language) imply a certain view of programming. The programmer had to *translate*: from the terms humans use to express a solution to a mathematical problem[4], to the series of machine instructions that corresponded to this solution.

The concept of 'human terms' did not so much pertain to the use of common, everyday language use. Rather it referred to formal mathematical notation systems. There was a long tradition of viewing mathematics as a language, the language nature, science and logic expressed themselves in (Nofre et al., 2014).

[4] In this thesis, the term 'mathematical problem' pertains to mathematical, arithmetical and logical problems in the widest sense of the word. I am not only talking about scientific and calculations, but also about jobs like managing databases with employee records, making a fun computer game, or calculating insurance premiums.

These systems allow humans to express procedures for solving mathematical problems in an abstract way. These abstract procedures are called *algorithms*. Algorithms can be translated to a computer. For instance, if one wants to know how much 5 percent of 2327 is, one has to divide the number 2327 by 100, and then multiply the result of this division by 5. This procedure can be generalized by replacing the specific numbers by letters. Then, it can be expressed in the common system of writing equations that many children learn at high school. The percentage algorithm will look like this:

y= (x/100) * p

This algorithm could be translated to a corresponding series of steps for a given processor. This involved many more steps than the steps expressed in the abstract mathematical formula. One had to specify instructions for loading the required numbers from memory to its processor registers (small bits of memory inside the processor), performing the needed operations, handling and storing intermediate results, and writing the resulting number from the processor register to the computer's memory.

Not only did the programmer have to specify each move of the processor, also (especially in the beginning), each machine was different. Computers had different sets of instructions available, different registers, and different memory sizes and arrangements. Part of the art of programming was dealing with the idiosyncrasies of the computer one was working on. The challenge lay in finding the most efficient way to run a general algorithm on a specific machine.

The translation therefore was a translation from the general to the specific, from the abstract world of mathematics to the inner workings of the concrete machine. Or, as quoted in an early paper by Hopper and Mauchly (1953)"It is (the programmer's) job to bridge the gap between the problem stated in terms of accounting procedures or mathematical equations and the specific coded instructions which control the computer system"(p.470).

But it was also a translation from the human programmer to his clever robot servant. Many early papers and books explicitly refer

to programmers as being human, to emphasize the contrast between the programmer and the machine.

Automatic Programming Systems

Very soon however, more and more aspects of this translation process became automated. These developments made it possible to move away from instructing the processor in its minutest details. These programs were referred to as automatic coding systems' of 'automatic programming systems' (I will discuss these developments to a greater detail later on in this chapter).

By the early fifties, there were programs that allowed the programmer to describe a calculation in very general terms, using a notation system akin to common mathematical notation. The computer took care of filling in the details. In our present view, the notation systems these programs used can be viewed as primitive programming languages.

But at that time, these notations were *not* viewed as languages (Nofre et al., 2014). When computer experts of that time talked about language, they either meant: the formal language of mathematics *or* the language of the machine. They did not conceive of autocoding systems and their notations as a language in itself. These systems were conceived of as an aide for the process, and nothing more.

This all has to do with the then-prevalent view of programming as mere translation, and not as conceptualization. Ideas about how to tackle a certain problem were formed in the abstract world of mathematical language. Only *after* having conceptualized an algorithm, the job of programming began: this algorithm had to be translated to the particular machine. These notation systems were seen as a part of the translation labor, and not as part of the conceptual labor that preceded the programming job. They did not seem to belong to 'the world of ideas' like formal mathematical notation systems.

Also, these notations were part of particular translation programs built for specific computers. This made it even less obvious to see them as abstract languages, like formal notation systems. These

languages could also not be classified under the category of machine languages. They were purely there for the convenience of the programmer. The computer did not use them as an internal language. Below all the automatic code functionality, computers still work by manipulating 'zeroes and ones'.

Therefore these notations fell outside of the two ways of using the word language in computing at that time. At that time, the concept of a programming language did not exist yet. Paradoxically while the first programming languages were being developed by computer scientists, they were not perceived of as languages. The idea of programming languages as a phenomenon did not materialize until the conceptual shift Nofre et al. (2014) talk about, took place. I will discuss this shift in a later section of this chapter.

The then prevalent view of programming also influenced how computer experts viewed working with automatic programming systems. Programming meant finding a translation between a mathematical solution to a problem and the corresponding machine code. Automatic programming systems (hence the name) were there to take this job from the programmer. Working with automatic programming systems was therefore not the same as programming, just like turning on the dishwasher is not the same as doing the dishes.

For instance, Grace Hopper expressed the hope that the automation of programming would enable the programmer to become a mathematician once again (Hopper, 1952) Brown and Carr III (1954) wanted to shift the burden of programming to the computer. The notations automatic programming systems used were called 'pseudo-code', because they were not part of the 'real' programming job.

Programming as Conceptualization

As said before (Nofre et al., 2014) have argued that modern-day computer experts and computer pioneers in the 1940's and early 1950's had a very different view of the activity of programming. While both groups use the word 'language' when talking about programming computers, they use this word in a very different way.

One of the important changes in the second half of the 1950's was that people started to use the term programming languages. The

notations used in automatic programming systems were further developed, and became to be seen as languages in themselves. This change was accompanied by a shifting idea about the activity of computer programming.

While programming in the early 1950 was seen as *translation*, modern-day computer experts see programming as *conceptualization*. Part of the programmer's job is to conceptualize a more or less elegant algorithm for some mathematical problem. This conceptualization is often done *in terms of the programming language the programmer is working with*. When the programmer comes up with an algorithm, it is formulated in terms of the input language for the computer, and can be directly used as input to the computer. No additional translation step is needed. Working out solutions for given problems, and conveying those solutions to the computer go hand in hand.

In the early 1950s these two worlds were still more or less separated from one another. First, the programmer thought of an algorithm in terms of the formal mathematical systems she learned in high school. Only *after* doing this, she would move on to the programming job: translating her ideas to corresponding computer commands. Since the second half of the 1950's these two worlds began to converge.

This had its effects on the status of working with an automatic coding system. At first people saw the activity of programming as a translation between abstract algorithm and concrete machine. Working with automatic programming systems was not seen as real programming, because one left the job of translation to the computer. But later, people started to see the activity of programming as the conceptualization of viable solutions. Because working with automatic coding systems also involves coming up with algorithms, people started to treat this activity as proper programming.

From Notations to Languages

The conversion between conceptualization and program writing is closely connected to another kind of change. Programming languages came to be seen as proper mathematical objects, on a par with other formal and logical systems. Like these formal systems, programming

languages are used to express mathematical problems and solutions. Once part of a translation process with mathematics as input, programming languages have become mathematics themselves.

These languages seem to have moved into the abstract, the world of ideas, away from the actual machines they were running on. This is closely connected to programming languages becoming *platform-independent* in the late 1950's.

Early automatic coding systems were still closely tied to one type of computer. Later systems could be used on more than one type of machine. For different types of computer, different translation programs were written. All programs however, accepted the same notation system. In this way one could write one program, using this notation system that could then be translated to different types of computer. Notation systems were no longer tied to one particular (type) of machine.

The focus shifted away from actual computers. At first, the technical challenge of automatic programming lay in getting it to do its translating job efficiently on the machine it was running on. Later however, developers of programming languages focused on the design of the language itself, with the translation to a particular computer being seen as a mere implementation detail. The concrete machine disappeared from the stage, leaving the notation system as an abstract entity, a language in itself.

A Brief History of Programming

In their article, Nofre et al. (2014) argue that our understanding of programming underwent an important shift. In the previous sections, I have discussed this shift. I have shown how programming was conceptualized in the early fifties. I have argued that computing, since then, has made an important conceptual shift. But how exactly, did this shift happen? Which kind of developments made it possible?

I have done a historical study, in which I researched the events that led to this conceptual shift. In the next section, I will discuss these events.

Automating the Programmer's Job

As shown in the previous section, instructing the computer directly, in its own internal machine language, was a very cumbersome job. One had to specify each move of the processor in its minutest detail. Also, input and instructions consisted of long strings of zeroes and ones, looking meaningless to the human, so next to cumbersome, this job was also error-prone. This led people to search for ways to automate parts of this process. These efforts gradually converged in the first automatic programming systems. Some of these systems are, in hindsight, seen as the first primitive programming languages. In the next sections, I will discuss these developments.

Mnemonic codes

Soon after the introduction of the first digital electronic computers at research institutes and universities, some groups working with these computers came up with the idea of a 'mnemonic code'. When designing a program, a programmer would write down codes like ADD STR or MUL, shorthand for add, store or multiply. When the concept program was finished, it was assumed that the next job for the programmer would be to translate it into zeroes and ones, before feeding it into the computer.

The director of the EDSAC project, (one of the world's first digital computers) asked one of his students, David Wheeler, to write a program that would read programs written in mnemonic code and convert them to the corresponding machine (Campbell-Kelly, Aspray, Esmenger, & Yost, 2014). The resulting program made it possible for programmers to insert programs written in mnemonic codes directly into the computer, instead of typing in endless series of zeroes and ones.

Subroutines

When doing calculations on a computer, often the same calculations are needed over and over again. In 1945, computer pioneers J. Presper Eckert and John Mauchly wrote about the idea of subsidiary routines. Instead of programming the same set of instructions multiple times, such a calculation could be programmed once and then stored in memory. Programmers could use this ready-made mini-program in their own programs, by instructing the computer to load it in to

memory and then execute it. The term subsidiary routine was abbreviated to 'subroutine' (Nijholt & van den Ende, 1994).

System programs

To start the system up, computers were equipped with system programs. These programs were at the base of everything the computer did. They were 'hard-wired' in the computer's circuits, and were loaded when the computer was switched on. These system programs would accept the programs prepared by programmers as input, convert them to actual electrical signals and make the processor run the program. In some cases, the hard-wired circuits would cause the master program itself to be loaded into memory, to run it as a normal program. This meant that certain places in memory could not be used by normal programs, for they would erase the master program. Any programmer working with a computer needed to know this before she would enter her program (Nijholt & van den Ende, 1994)

Symbolic addressing

Because of issues like this, programmers had to be aware of the computer's memory, which locations could be used and which could not. Subroutines made this task even more complicated. Like the master program, subroutines also needed a place in memory, for the subroutine instructions themselves and for storing the subroutine's intermediate results. Therefore, the master programs were assigned a new responsibility. They were to manage the computer's memory. Instead of letting the programmer specify specific memory addresses, the master program allocated these memory addresses. The programmer only had to refer to a 'symbolic' address while the master program kept track of the 'real' address of this value. This was called relative or symbolic addressing. Symbolic addressing made the use of subroutines less complicated (Nijholt & van den Ende, 1994).

Generators

The idea of letting computers do their own 'housekeeping' was developed further in 'generator' programs. Programmer Betty Holberton wrote the first generator program (Hopper, 1981). Her Sort-Merge Generator was designed to merge two files. It accepted the specifications of the targeted files as input, and used that input to produce a series of machine instructions. These

instructions would merge the files in the desired way. The generator was designed in such a way that the resulting program would handle the allocation of memory locations also correctly.

SHORT CODE

The idea of using subroutines was also carried further, in the SHORT CODE program. This program was suggested by John Mauchly, and written by R Logan, W Schmitt and A Tonik (Sammet, 1969). Its goal was to give the BINAC computer extra functionally. The limited hardware of that time could not process floating point numbers. The conversion had to be done arithmetically. This program made that conversion easier.

The program also offered a coding system, which the programmer could use to specify equations with. Instead of guiding the processor step-by-step through a series of operations, one could write down a series of codes, each standing for a mathematical symbol or a number. Together these codes would form an equation which the program would recognize. The program would then call in a series of subroutines, which delivered the desired result.

This was different from what Wheeler had done. Mnemonic codes, like the Wheeler system, were easier to remember than strings of bits, but these notations still corresponded one-to-one to processor instructions. This meant that one still had to instruct a processor into the minutest details of the calculation. SHORT CODE made a move away from these instructions, making it possible to instruct the BINAC on a more general level. SHORT CODE was later reprogrammed to be used on another computer, the UNIVAC.

The A-0 compiler

In 1951, Grace Hopper received an assignment to build a number of mathematical subroutines for the UNIVAC computer. The routines were to be standardized in such a way that everybody could use them. Hopper decided to write a program that would allow the user to list the subroutines she wanted to use. The subroutines were assigned unique ‘call words’. The user typed in a series of call words for the subroutines she wanted to use. After each call word she would enter some specifications needed for that particular calculation. The program took in these lists and compiled a ready-made program

from the available subroutines. This program was called the A-0 compiler.

The reason it got called a compiler was that each subroutine was given a 'call word' because the subroutines were in a library, and when you pull stuff out of a library you compile things. It's as easy as that. (Hopper, 1981a, p. 10)

This program produced its own machine language programs, and took care of the memory management task. The A-0 compiler was based on a different principle than SHORT CODE. SHORT CODE would read instructions one by one, decode them and for each decoded instruction, call in the corresponding subroutine. The A-0 compiler would read a complete coded program, like the Sort-Merge generator did. Only after it finished 'reading' this entire program, it would start compiling a corresponding machine language program.

Automatic Programming

In 1953 Grace Hopper and John Mauchly wrote a paper in which they discussed several ways in which the computer could assist the programmer with her job. In this paper they defined the concept of a compiler as a program that creates a complete program, instead of translating and linking subroutines at each step (Hopper, 1981; Hopper & Mauchly, 1953).

The difficulty and error-proneness of programming was at that time an important bottleneck for computing (Hopper & Mauchly, 1953). There was a shortage of trained programmers, and "the cost of programmers associated with a computer center was usually at least as great as the cost of the computer center itself"(Backus, 1981, p. 26). But still, the idea of automating machine programming was not easily accepted. "We found that we had to change from being research and development people and turn ourselves into salesmen, and get out and sell the idea of writing programs this way to customers" (Hopper, 1981, p. 14).

The A-2 compiler

After some time working with the A-0 compiler, "the long and cumbersome way of writing the input specifications to the A-0 compiler were becoming apparent and it seemed much better to have a

shorter way of writing that stuff" (Hopper, 1981a, p. 12). Therefore, Hopper and her team devised a new scheme for coding input, which was more akin to the input format of the SHORT CODE system. Like SHORT CODE, this compiler also allowed for instruction on a more abstract level than machine instructions.

Instead of listing series of call words, the new compiler made use of three-digit 'code words'. The programmer could specify an operation, two input variables and one result variable, coded in these words. For instance, a typical instruction would be: ADD 00A 00B 00C which would mean: Add the values stored in A and B and store the result in C. The computer was to recognize the coded programs and to generate the corresponding machine programs (Hopper, 1981; Sammet, 1969). In 1953, Hopper and her team augmented the A0 compiler with new code that enabled the compiler to recognize this new input scheme. The resulting compiler was called A2.

Synthetic computers

In the early 1950's more automatic programming systems came into use. These programs aimed to simplify the programmer's job. Most of these systems were aimed at overcoming the hardware limitations of that time, such as the lack of floating point functionality. These systems tried to overcome that problem by letting their program simulate the behavior of a virtual, more advanced processor, called a synthetic computer. This virtual processor could handle floating point numbers, and was equipped with extra (virtual) registers. It was therefore easier to program.

Programmers wrote a set of instructions for this virtual processor. The program would then read these virtual processor instructions and translate those into instructions for the real processor (including the arithmetic conversion from floating to fixed point numbers). These programs made programming somewhat easier, but this came with a price. They typically slowed the computer down with a factor of five to ten (Backus, 1981). According to Backus, experience with slow and inefficient automatic programming systems led many programmers to believe that machine language programming could not be automated.

These automatic programming systems typically did not abstract away from machine language. It is true that the programmer gave her instructions in another format than the computer's actual machine code. They were, like in the SHORT CODE system, using a pseudo-code. But that did not change the fact that they still were instructing every move of a processor, albeit a virtual one.

Algebraic Translators

While most automatic programming systems at that time did not abstract away from processor instructions, there were some systems that did, like the A-2 compiler. Another example is the algebraic translator', developed for the MIT Whirlwind, in 1954 by J.H Laning and N Zierler. The notation used by this system resembled 'natural' mathematical notation. Unlike SHORT CODE or the A2 compiler, the programmer could write numbers, operators and letters directly, without having to translate them in some digit code.

Statements looked like this:

```
c = 0,5 / d
```

or

```
a = c + d
```

These systems were referred to as 'algebraic' because their input notations aimed to resemble formal algebraic notation as closely as possible. Some of these systems made use of a compiler, and therefore resemble modern-day programming languages, (although they were not platform-independent yet.) The notations these systems used are in hindsight, seen as the first 'real' programming languages.

But at that time, the concept of a programming language did not exist yet. Algebraic translators were first of all a class of automatic coding programs. The conference proceedings of the 1954 MIT summer sessions also show that, at that time, many people did not seem to notice the importance of moving away from specific processor instructions.

Laning and Zierler's algebraic translator gets only three pages in the 28-page document. The rest of the document is devoted two other

automatic programming systems, that are based on the principle of a virtual processor. (*Proceedings Symposium on Automatic Programming Digital Computers*, 1954). In the introduction of the text several techniques for simplifying programming are discussed. These include mnemonic codes, subroutines and relative addressing. But to a functional algebraic translation system (like Laning and Zierler's) these techniques are mere implementation details, instead of competing alternatives.

FORTRAN: Between Programming Language and Automatic Programming System

Between 1950 and 1955 the use of computers became more wide-spread. Next to the military and research institutes, insurance companies, universities and government agencies also purchased computers to help them with their work. The first production line computers, the Remington Rand UNIVAC and the IBM 701 became available (Nofre et al., 2014). In 1954 users of the 701 formed a special user association, called SHARE in order to share ideas and knowledge (Nijholt & van den Ende, 1994). These new users were plagued by all sorts of problems. Because in the early 1950's programming was still so very close to the inner working of the processor, even relatively simple tasks took a lot of programmer effort. Also, programs written for one computer could not be transported to another type of computer. This would make upgrading to a new system very costly, since all the programs had to be re-written for the new machine (Nofre et al., 2014).

This of course was a problem for the manufacturers of those machines. IBM tried to remedy this by encouraging the exchange of programs between their customers. Sharing knowledge was hoped to prevent duplication of work. Because the 701 was a production-line machine, programs written on a 701, could in principle be run on other 701's. If a user knew of somebody else who had already written a program for a task she wanted to accomplish, she could use that person for that program, instead of writing it herself. Still, their efforts did not prevent a significant duplication of work from 701 users.

John Backus, a programmer at IBM tried to tackle the problem from a different angle. Perhaps, instead of preventing duplication of effort, perhaps it would make sense to tackle the difficulty of programming itself. Backus was thinking among the same line as Hopper, Laning and Zierler, although at that time, he did not know of Laning and Zierler's work (Backus, 1981). His solution was to write an automatic programming system that abstracted away from specific processor instructions, (or in other words an algebraic translator). This program was to be written for IBM's newest model, the IBM 704. The program would be called FORTRAN, (FORmula TRANslator).

His superiors said yes to the program, but the company did not expect serious results from the project. They thought of it as a research project. At that time, many people thought that machine language could not be automated (Backus, 1981; Campbell-Kelly et al., 2014).The project was located next to the elevator room, and was not part of the product plan for the 704.

Backus and his team started working on the project in the beginning of 1954 and were finished in April 1957. Together, FORTRAN took 18 man-years to complete (Nijholt & van den Ende, 1994). Although more ambitious and complex than previous algebraic translators, this program did not seem to differ fundamentally from these earlier systems. Like earlier systems, FORTRAN was first and foremost a computer program, with a corresponding notation system. Most of the effort went into getting the program to do its job in an efficient way. Designing the notation system only came second (Backus, 1981). Also, like earlier algebraic translation-systems this program was designed for a specific machine: the IBM 704.

But unlike the A-2 compiler and the algebraic translator, FORTRAN was built for an assembly-line computer. This meant that even though it was not platform independent, it still could run on many different machines. FORTRAN turned out to work very efficient. Programs were developed in much shorter times, days instead of weeks (Campbell-Kelly et al., 2014). And because there were many 704's in use, many people could profit from FORTRAN. This made the system a

huge success. For the first time, an algebraic automatic programming system was widely used.

Although Backus never had meant his system to be platform-independent (Backus, 1981) the success of the FORTRAN system turned out to reach beyond the 704. At that time, another algebraic automatic programming system, the IT system, had been successfully adapted to work on multiple machines. Inspired by this work, IBM designed a system to adapt FORTRAN to one of its other models, the IBM 650. A second version of FORTRAN was made available on several other IBM models. In the early sixties, FORTRAN was also used on non-IBM computers (de Beer, 2006; Nofre et al., 2014).

Backus seemed mainly interested in abstracting away from specific processor instructions. But the success of FORTRAN was important to a movement that searched for another kind of abstraction. At that time, many people thought that the future of computing lay in universality. Regardless of the type of computer, computers should understand the same language, so that programs could be exchanged between different computers. The wide adaptation of FORTRAN on different machines showed them that this was technically feasible. This, inspired efforts to create a universal computer language. One attempt to build such a language would be a catalyst to the conceptual shift that created our modern idea of a programming language.

The Search for Universality

In their article (Nofre et al., 2014) explain how the proliferation of computing installations, and the growing role of industry in computer building, brought about a search for universality in computing. Before the 1950's, computing was mostly confined to a few military research institutes and universities. The computers they used were often built by these organizations themselves. In the early 1950's, businesses and government agencies started to use computers also. Building computers became a commercial activity.

All these different actors brought their own interests with them. Many organizations working with computers wanted to control the costs associated with programming computers. They sought for ways to

port computer programs from one type of machine to another. Computer manufacturers were also aware of this problem, and encouraged collaboration and knowledge sharing between users. Computer users shared their knowledge in user groups. Scientists also supported initiatives to create portable computer programs, because they felt that a greater interchangeability of computer programs would benefit the exchange of computer knowledge.

Despite the fact that computer manufacturers generally supported the exchange of knowledge, scientists saw the new commercial interests in computing as a threat to the free exchange of information. Up until then, computer centers ran on government and military funding. The military tended to encourage collaboration and the free exchange of knowledge. It was feared that computer manufacturers would try to protect their business secrets and replace this open atmosphere with secrecy.

In the early 1950's, different scientists came up with the idea of a universal computing language. This would be a language that was not connected to any particular machine. For instance, Gorn (1954) discusses the possibility of a universal code that was to be "more or less independent of the machine" (p.75). Such a language was hoped to stop the treat of computer manufacturers becoming too powerful:

Nevertheless, as an alternative of the commercial capture of the computer and data processing field by one make of machine, or arbitrary ruling on machine specifications by government fiat, one now has the interesting possibility of a common, universal, external language arrived at by mutual agreement and persuasion, which can be matched to the internal structure of numerous computers by 'black boxes' which translate and generate the required computer internal programs. (Brown and Carr, 1952, pp. 89-90)

Note the use of the term language, instead of 'notation system'. The proposed universal language was not part of an automatic programming system. Rather, it was seen as a language in itself, a free-standing entity, not connected to any computer in particular. These are the first signs of a changing meaning of the word 'language' in

computing. When in previous times, computer experts talked about language, they either meant machine language *or* abstract, mathematical languages. Now, the notations that simplified coding were also seen as languages.

The success of FORTRAN, which had an advanced notation system, and could be ported to different machines, showed that, such a universal language was technically feasible. This fueled the effort to develop such a universal language. In the early sixties, different initiatives were taken to develop a universal computing language. One of these initiatives was COBOL (Common Business Oriented Language), an initiative from the Ministry of Defense. This was an automatic programming system akin to FORTRAN, but designed for use in businesses. The idea was that if all organizations would use COBOL for their data-processing, they could exchange programs. The first COBOL compilers were built in 1962. The ministry of Defense required firms who wanted to do business with them to use COBOL (Nijholt & van den Ende, 1994).

Another initiative to achieve portability between different systems was UNCOL (Universal Computer Oriented Language). This initiative took a different approach than COBOL. It did not depend on the willingness of all users to adopt one single language system. Instead it accepted the proliferation of languages as a given (Nofre et al., 2014).

The idea was to develop a 'universal' machine language, UNCOL. This language was more or less like the synthetic computer systems described earlier. Like those systems, the language described every move of a virtual processor. Synthetic computers were developed to overcome the hardware limitations of their real processors. The synthetic processor was therefore more or less independent of the particular processor running it. This gave it a kind of universality. The developers of UNCOL hoped that that universality would be the key to a universal computer language.

Unlike synthetic computer style languages the UNCOL language was not meant for human programmers. Instead, it was meant as an intermediate step in a bigger translation process. This process

combined two compiling steps. First, it would take a program in a higher level language, like COBOL or FORTAN. This program would be translated into UNCOL. This process would be the same on every machine. After that, a machine dependent compiler would take the UNCOL program and translate that into instructions for its own processor. It was hoped that this system would enable programs in different languages to run on different computers, enabling the free exchange of programs. But the development of such a system turned out to be harder than expected, and the development of UNCOL was stopped in 1962 (Nofre et al., 2014).

FORTRAN itself, despite the portability and easy programming it offered, was not considered a candidate for this computing lingua franca. This was done intentionally. As said before, many people feared the growing influence of commercial computer builders. If IBM's FORTRAN were to become the universal computer language, it was feared this would further strengthen its already dominant position (Rosen, 1967).

ALGOL

At the same time, in Europe, academics also made efforts to develop a universal computing language. In 1957, a group of European academics sent a letter to the president of the largest computer association in the United States, the ACM. They proposed collaboration between US and European computer experts, to create a common formula language (Nofre et al., 2014).The ACM agreed to this collaboration. The language was to be called International Algebraic Language (IAL). Later, the language was renamed to ALGOL (ALGOrithmic LANguage). It was this language that brought about the conceptual shift that is at the basis of our modern understanding of programming.

This initiative differed from the initiatives before it. The academics saw their new language as a scientific project. Of course, a universal computing language would be a great aid for scientists, for it would facilitate the exchange of information. But the scientific ambitions of this project stretched beyond this. The new language was to turn computing itself into a subject of scientific study.

The language was designed to describe algorithms, the general solution procedures behind the computations carried out by computers. Describing these algorithms in a clear and concise way would make it easier to study these algorithms. This meant that the language needs to make use of a precise and elegant notation system. But *control statements* also mattered. These statements describe the overall structure of the computation that is performed. They designate in which order the different steps of the computation are to be carried out, and the conditions under which different actions are to be carried out. The language needed to have a set of control statements that would give the language a clear structure. These statements should not depend on the idiosyncrasies of the machine the language was originally developed for (many programming languages of that time suffered from this). A clear set of control statements would allow scientist to reason in an abstract way about the structure of computation.

The idea was that computing in itself was a topic worth of scientific study, and the new language would make a clear description of computing possible. Next to this, of course it was also hoped that this language would become the computing lingua franca that would make the easy exchange of programs possible.

Language, True Language and Nothing but a Language

Because from the beginning ALGOL's developers conceived of their project as an abstract language, they took different approach from the approach Backus followed with the development of FORTRAN (and all the algebraic translators before FORTRAN). FORTRAN had started as a program for the IBM 704. Getting this compiler program to run efficiently had been the biggest challenge for developers. The corresponding formula notation only came second. The developers of ALGOL did the exact reverse. Language design came first. Getting the compiler to work not even came second; they did not even bother to build a compiler at all. ALGOL was not an algebraic translating program. Like natural and formal languages, it was a language, but also *nothing but* a language.

From the standpoint of universality, the choice not to provide a corresponding compiler made sense. The designer's job was to provide

a universal language, a tool that would enable different computers to communicate with each other. Getting this tool to work in specific situations was not the task of the tool designer, but the task of the workman using the tool.

In the beginning however, there were some problems with this no-compiler approach. Although the designers tried to describe their language in a clear and concise way, they could not prevent some subtle ambiguities creeping in. This led to different interpretations of the language. This in turn resulted in different legal ALGOL compilers. The language that was to be *the* single universal programming language, now knew a variety of local dialects! No good news for its developers, so they decided to make some improvements (Nofre et al., 2014).

To avoid different interpretations the syntax and semantics of the new language were precisely defined. John Backus (who also was part of the ALGOL team) got the idea to describe the syntax by means of an 'artificial grammar', a formal notation for describing the rules according to which legal statements in the language should be formed. This notation, called BNF (Backus Naur Form) was analogue to the notation used by logician Emil Post (Nijholt & van den Ende, 1994). The new language was called ALGOL '60.

This new language, with its explicit emphasis on abstraction, was the final step in a long process of abstracting away from the inner workings of the actual machine. Nofre et al. (2014) argue that the real, physical computer became increasingly *black-boxed*, hidden behind a set of abstract principles.

This process started with the first automated coding systems and was accelerated by the promises of universal, machine-independent languages. Automated coding systems such as SHORT CODE or Laning and Zierler's algebraic translator, enabled the programmer to leave the cumbersome details of programming to the computer. The machine-independent language FORTRAN enabled the same program to run on multiple machines, separating computer programs from specific machines. And ALGOL went yet another step further, by viewing their

programming language as an abstract, mathematical language, detached from any actual translation program for that language.

Closing the black box: A new understanding of computation

These developments detached the notion of a computer program from the notion of an actual computing process. In the next section, I will argue that this separation led to two important realizations. These two realizations provided the cornerstones for a new, scientific understanding of computation.

I will show how the disconnection between computer and computer program could bring about these new insights. Then, I will show how these new insights enabled a new way of thinking about computation, providing a new paradigm for the scientific study of computation.

Cornerstone one: a subject to study

The first of these two key insights was brought about by a shift in people's understanding of computer programs. Initially, people's understanding of computer programs was intimately tied to the specific way actual machines carried out these programs. But technical developments, such as automatic programming and universal languages, caused the notion of a computer program to become disconnected from the specific type of machine it was running on.

These developments changed people's understanding of computer programs. They started to approach programming on a higher level of abstraction, independent of the specific machine the program would run on. Over time, people started to realize that they had developed an abstract conception of computer programs. They now understood computer programs in terms of *algorithms*.

This abstract conception of computer programs became a topic of interest to many people. Viewed as actual machine processes, programs are just one particular solution on one particular machine. But when understood as general algorithms, they illustrate how entire classes of problems can be solved, providing insight in the principles of algorithmic (machine) calculation itself. Therefore, these programs can be understood as instances of algorithmic computation. This was an interesting phenomenon in itself, beyond

the mere technicalities of specific computers. These programs started to become recognized as a topic worth of scientific study.

This was the first of the two key insights, from which the field of computer science would emerge. It provided the subject for this field to study. Computer science would become the study of *algorithmic computation*.

Cornerstone two: a way to understand

The second of these two key insights was brought about by a shift in people's understanding of algorithmic notation formats, like the algebraic notation that was used in Laning and Zierler's algebraic translator. Like computer programs, people's understanding of these algorithmic notation systems was initially closely tied to a specific machine.

These notation formats were able to describe computing on an abstract level, independent of any specific machine. But although these notation systems were machine-independent, their initial goal was anything but. They were built for a specific machine. They were part of a program that could translate this abstract, algebraic description into the internal language of this machine.

For instance, Laning and Zierler's algebraic translator was built for the Whirlwind computer. And although FORTRAN became the first machine-independent notation system, it was initially designed to work on just one system, the IBM 704. Therefore, these notation languages were still understood as being tied to a specific machine. But when the automation of machine programming and the ideals of universality disconnected the understanding of computing from the actual machine, the initial goal of these notation formats became less prominent.

Initially, constructing machine language programs was viewed as the final objective of the programming process. But gradually, this activity became a less important step in the process. Eventually,

this activity would become nothing more than an implementation detail, completely handed over to the computer[5].

When the construction of actual machine programs became less prominent, other aspects of the programming activity became *more* prominent. Drafting a mathematical sketch of the general algorithm had always been a preparation stage for the 'real' job. When that job was taken over by the computer, the algorithmic sketch gradually became to be viewed as 'the real job'.

This changed people's perception of these notation formats. Because the goal of building machine programs moved to the background, their function in a machine language translation system became less relevant to people. Describing the general algorithmic structure of the program had now become the main activity of programming. Therefore, their function as a general description format for algorithmic computation became all the more relevant to people. Gradually, people started to focus on the notation itself, while the rest of the translation program moved to the background.

This led to the second important insight. The activity of programming now consisted of describing general algorithmic procedures to the computer. These algorithmic notation formats provided a precise, mathematical notation to describe these procedures in. Therefore, people started to realize that these notation systems could be viewed as a formal language for describing computation.

At this point, these notation systems stopped viewed as being a part of a translation program and became to be viewed a languages in themselves. They were seen as the formal, mathematical language of algorithmic computation. According to Nofre et al. (2014) these notations had become abstract structures. Therefore, people no longer understood them as connected to any specific program[6].

[5] In one of his famous EWD writings, Edsger Dijkstra (1989) beautifully reflected on this development, 'After all, it is no longer the purpose of programs to instruct our machines; these days, it is the purpose of machines to execute our programs' (p. 5).

[6] The idea of describing computation with a formal language was not entirely new. The builder of one of the earliest modern computers, Konrad

This was the second important insight that enabled the field of computer science to emerge. As I have explained above, the first key insight provided the field with its topic of study: *algorithmic computation*. The second key insight provided an idea about *how to understand* this form of computation.

People realized that programming languages could be used to study these algorithms (Nofre et al., 2014; Priestley, 2011). Later, the discipline also made use of other mathematical descriptions, to capture certain specific aspects of computing. But programming languages provided the first powerful means to describe this form of computation.

These ideas were the foundation for the new paradigm, around which the field of computer science emerged.

The emergence of a paradigm

The ALGOL language played an important role in the establishment of the new paradigm of computer science. It was designed to be a description tool for the study of algorithmic computation. But it also served another function. It helped to establish the new paradigm, by 'communicating' the ideas of this new paradigm.

The design of this language expressed an idea about what kind of subject was studied here. According to the design of this language, the study of computing is the study of the abstract. ALGOL was designed to resemble the artificial languages of formal language theory, developed by logicians in the 1930's. Because it was meant to be a purely abstract language, not an automatic coding system, it had no corresponding translation program. This design emphasized the abstract nature of the subject, clearly distinguishing it from the physical computer.

Previously, studying computers had pertained to: studying specific machines. It was an engineer's job to gather knowledge of these

---

Zuse, already developed a programming language for the description of computer programs in 1945 (Nijholt & van den Ende, 1994). His views however, did not receive much attention initially. This was probably because at that time, people's idea of programming was still connected to programming a specific machine. Therefore, they saw no use in describing programming in abstract terms.

machines. This was *applied* knowledge, mainly of interest to the engineer, who sought to improve the machines (and only indirectly of value to scientist and the rest of the world, who of course, did benefit from better computers).

The academics behind ALGOL wanted to shift attention from the machines themselves to the principles *behind* those machines. These principles were abstract principles, independent of any existing computer. Therefore, they were a topic worth of scientific study in itself.

By doing this, they claimed a distinct scientific status for computing. From then on, computer science was established as a scientific field, with a corresponding set of practices, terminology, textbooks, journals and all the other things that come with a mature scientific field (Nofre et al., 2014). These academics had succeeded in their scientific ambitions for their new language.

As said before, this conceptual shift moved programming notations into the domain of formal mathematical languages. This made that the properties of programming languages could be studied with the conceptual tools that were used study other formal systems. For instance, Ginsberg and Rice proved that the notation BNF was equivalent to a family of formal grammars described in a paper by Noam Chomsky (Chomsky, 1956; de Beer, 2006; Ginsburg & Rice, 1962).This gave rise to a whole new body of literature that connected programming languages with formal linguistics, logic and meta-mathematics (the study of properties of mathematical deduction systems).

In this chapter, I described how the paradigm of computer science emerged. Using the work of Nofre et al. (2014), I have shown how in the 1950's, people's understanding of computer programs changed. This conceptual shift provided the two cornerstones for the development of understanding in computer science.

But what kind of understanding is this? What does it mean to seek understanding of algorithmic computation, as executed by computer programs? How can a programming language help in the development of this understanding?

In the next chapter, I will analyze this form of understanding. I will relate it to the understanding developed in mathematics and the understanding developed in the field of physics. This will allow me to show to what this understanding is similar to these traditional forms of understanding and to what extent it is different.

References

Backus, J. (1981). Paper: The History of Fortran I,II and III. In R. L. Wexelblat (Ed.), *History of Programming Languages*. Pennsylvania: Academic Press.

Berkeley, E., C. (1949). Giant Brains or Machines That Think. New York: John Wiley & sons, inc.

Brown, J., & Carr III, J. (1954). *Automatic Programming and its Development on the MIDAC.* Paper presented at the Symposium on Automatic Programming for Digital Computers, Washington, DC.

Burks, A. W., Goldstine, H. H., & von Neumann, J. (1946). Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. *The Institute for Advanced Study*.

Campbell-Kelly, M., Aspray, W., Esmenger, N., & Yost, J. R. (2014). *Computer: a History of the Information Machine*. Boulder, Colorado: Westvieuw Press.

Campbell, R. V. (1952). *Evolution of Automatic Computation.* Paper presented at the Proceedings of the 1952 ACM national meeting (Pittsburgh).

Carr III, J. W. (1952). *Progress of the whirlwind computer towards an automatic programming procedure.* Paper presented at the Proceedings of the 1952 ACM national meeting (Pittsburgh).

Chomsky, N. (1956). Three models for the description of language. *Information Theory, IRE Transactions on, 2*(3), 113-124.

de Beer, H. (2006). *The history of the ALGOL effort.* Masters Thesis, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science.

Ginsburg, S., & Rice, H. G. (1962). Two families of languages related to ALGOL. *Journal of the ACM (JACM), 9*(3), 350-371.

Gorn, S. (1954). Planning Universal Semi-automatic Coding *Symposium on Automatic Programming for Digital Computers* (pp. 74-83). Washinton D.C: Navy Mathematical Computing Advisory Panel

Hopper, G. M. (1952). *The Education of a Computer.* Paper presented at the Proceedings of the 1952 ACM national meeting (Pittsburgh).

Hopper, G. M. (1981). Keynote Adress. In R. L. Wexelblat (Ed.), *History of Programming Languages* Pennsylvania: Academic Press.

Hopper, G. M., & Mauchly, J. W. (1953). Influence of Programming Techniques on the Design of Computers. *Proceedings of the IRE, 41*(10), 1250-1254.

Laning Jr, J., & Zierler, N. (1954). A program for translation of mathematical equations for Whirlwind I. Engineering Memorandum E-364: MIT Instrumentation Laboratory.

Levin, J. H. (1952). *Construction and use of subroutines for the SEAC.* Paper presented at the Proceedings of the 1952 ACM national meeting (Pittsburgh).

Nijholt, A., & van den Ende, J. (1994). De Geschiedenis van de Rekenkunst: Van Kerfstok tot Computer (The History of Computing: From Tally-Stick to Computer).

Nofre, D., Priestley, M., & Alberts, G. (2014). When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960. *Technology and culture, 55*(1), 40-75.

Priestley, M. (2011). *A science of operations: machines, logic and the invention of programming*: Springer Science & Business Media.

Proceedings Symposium on Automatic Programming Digital Computers. (1954, 13-14 May 1954). Paper presented at the Symposium on Automatic Programming Digital Computers, Washington DC

Rosen, S. (1967). Programming Systems and Languages, A Historical Survey. In S. Rosen (Ed.), *Programming Systems and Languages*. London: McGraw-Hill.

Sammet, J. E. (1969). *Programming languages: History and fundamentals*. New Jersey: Prentice-Hall, Inc.

Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, 433-460.

Chapter 7: Analyzing Understanding in Computer Science

Introduction

To describe the development of understanding in computer science, I conducted a literature study about the history of computer science. By showing how this form of seeking understanding emerged, I was able to describe the specific way computer science seeks understanding. In my literature study, I established that computer science, as a scientific discipline, seeks an understanding of algorithmic computation. The field does this by describing this computation with programming languages and other mathematical models.

In this chapter, I will relate this form of seeking understanding to the more traditional forms of understanding. I will start with a short discussion of these forms of understanding. Then, I will explain why computer science does not seem to fit in with these forms of understanding. I will present my view about the understanding developed by computer science and provide a short outline for the rest of this chapter. Then, I will move on to the main part of my analysis.

Different kinds of scientific understanding

The chapters about understanding in physics and understanding in mathematics show that there are different kinds of scientific understanding. Scientists can seek an understanding of the nature of our physical world. Or they can seek to develop an understanding of our basic concept of structure. These are different kinds of understandings of different 'things'. Therefore, each form of understanding is developed in a different way.

Understanding the physical world versus understanding structure

Understanding of the physical world is developed through finding causal explanations for physical phenomena. These phenomena occur to us without an observable cause. We strongly feel that all things must have a cause in order to occur, but we are not able to see these causes. Therefore, we assume that these phenomena must be caused by something invisible. To explain why these phenomena

occurred, we therefore need to discover what sort of hidden thing has caused them.

Because these causes are unobservable to us, their nature can only be inferred from the phenomena that result from them. Therefore, in order to draw inferences about these hidden causes, we need to observe these physical phenomena. We have to check whether the actual behavior of a phenomenon matches the ideas we have formed about it. An understanding of a physical world therefore, is an empirical understanding. It is built on actual experiences with actual physical phenomena.

We can also develop our understanding of structure. We perceive structure everywhere around us. Therefore, this structure appears to be a fundamental part of our reality. This would imply that our understanding of structure is an understanding of reality. But this is not the case. The structure we perceive in the world is not a part of objective reality.

Our experience of structure is created by our own brains. Our brains use an innate, pre-existing notion of structure, to interpret the inputs they receive. Through this active process of interpretation, our brains make us experience the world as a world filled with structure. This enables us to make sense of this reality.

Because our concept of structure is not a part of objective reality, it cannot be found in this reality. Therefore, our understanding of this concept is not developed by seeking it in objective reality. It is developed through *analysis*. This means that we carefully examine our pre-existing notion of structure, in order to reason about this notion.

Theories in mathematics provide formal descriptions of our concept of structure. Such a precise description enables us to reason more precisely about our intuitive ideas of structure. This enables us to further develop our understanding of these ideas.

Why computer science doesn't seem to fit in these categories

Most fields in natural science can be understood as seeking one of these forms of understanding. The classical experimental sciences,

such as chemistry, biology and physics, seek an empirical understanding of actual things. The subfields of mathematics use analysis to seek an understanding of our abstract concept of structure.

But computer science is strange. Somehow, it seems difficult to see where computer science would 'fit' within this picture. It is different from abstract mathematics. But it doesn't seem to fit in with the classical experimental sciences either.

I believe that it is difficult to categorize computer science, because the field has developed its own, specific way of seeking understanding. This manner of developing understanding doesn't fit in with either of those categories, because it actually incorporates elements of both. Computer science studies algorithmic computation, which, as I will show, is an understanding of structure. But on the other hand, the development of this understanding involved 'empirical' experiences with actual computers.

In this chapter, I will explain how computer science can develop an understanding of structure through experience with physical objects. First, I will compare the field of computer science with the field of mathematics. I will argue that computer science is similar to the field of mathematics, because both fields seek to develop an understanding of structure.

Then, I will discuss the different ways in which these fields develop an understanding of structure. The field of mathematics develops an understanding of structure by creating formal descriptions of this structure. But computer science also uses programming languages to describe structure. I will explain how programming languages help in developing an understanding of structure.

Next, I will explain why the empirical understanding developed in computer science is not the same as the understanding developed in natural science. Then, I will show that physical objects do not only help us to understand the physical world. They can also help us to develop our understanding of structure.

Finally, I will explain why computer science was able to develop such a different way of understanding structure. I will show why the field's specific subject allows physical objects to play a powerful role in developing an understanding of this subject.

Computer science and mathematics: Why an understanding of algorithmic computation is an understanding of structure

In the previous chapter, I have established that computer science seeks an understanding of algorithmic computation. But what sort of phenomenon is this? Can we understand 'algorithmic computation' as a physical phenomenon? Or is it a part of our intuitive concept of structure? Does that mean that computer science, like mathematics, is an abstract study of structure? Or does it, like physics, also study a part of the physical world?

In the next section, I will argue that algorithmic computation, as being studied by modern theoretical computer science, is part of our concept of structure. Therefore, in respect to the subject it studies, the field of computer science is therefore more akin to mathematics than it is to physics.

According to Nofre, Priestley, and Alberts (2014) computation only became a topic of scientific study after people started to understand computer programs in more abstract terms. This implies that the field does not seek to understand computation as a physical phenomenon.

Scientists and engineers had been developing and building physical computers for years. But they never reported anything worthy of scientific investigation. People only started to notice interesting phenomena when they started to view computer programs as abstract objects. This means that the phenomena computer science seeks to understand were not found within in the physical computer. Therefore, it is unlikely that these phenomena were connected to the physical workings of the computer. The phenomena that computer science seeks to understand are abstract phenomena.

Further developments in this field only confirm this view. In my history study, I discuss how the physical computer got 'black-boxed´. All references to actual, physical implementations were to

be avoided. Computing processes had to be described as abstractly as possible. These descriptions used the formal language of mathematics (Nofre et al., 2014). The newborn field wanted to make it absolutely clear that it was studying an abstract idea of structure, instead of a physical device.

Of course, computer science is about understanding actual, physical computing too. We seek an understanding of abstract computing processes because we want to understand and improve actual computing processes. Therefore, one could make the argument that in the end, this field *does* seek to understand physical computing processes.

But that doesn't mean that this field is developing a 'physical' type of understanding. That would be to confuse the subject of the field with the goals of the field. Understanding actual computing is one of the field's end goals. But the understanding developed in order to *reach* that goal is an abstract understanding of structure. Therefore, computer science, like mathematics, seeks an understanding of structure.

Describing structure: formal theories versus programming languages

In the previous section, I have shown that the phenomenon studied by computer science is a part of our concept of structure. Therefore, computer science seeks to develop an understanding of structure, just like the field of mathematics does.

But there is also an important difference between computer science and mathematics. I have shown that mathematics understands structure by creating formal descriptions of this structure. Computer science, however, also uses another kind of description. Next to using 'traditional' mathematical descriptions, this discipline also makes use of programming languages.

Why does computer science often prefer such a different kind of model? How are these languages related to more traditional, mathematical descriptions of structure? And how can programming languages describe the structure of algorithmic computation? In the next section, I will try to answer these questions.

First, I will explain how traditional mathematical descriptions help us understand algorithmic computation. Then, I will explain why programming languages are so fruitful for understanding algorithmic computation.

How traditional mathematical descriptions help us to understand algorithmic computation

According to Nofre et al. (2014) the field of computing was able to develop into a scientific discipline after people started to think of computer programs as abstract algorithms. The notion of the algorithm as an abstract mathematical structure however, predates the modern computer. Before the introduction of the modern computer, algorithmic procedures were already carried out by humans. Humans still do this. A child that is doing long division in school is carrying out an algorithm.

Mathematicians tried to understand these procedures as abstract structures. In the second half of the 1930's they developed the first mathematical descriptions of algorithmic computation. In the next paragraphs, I will show how the creation of this description enabled mathematicians to develop an understanding of algorithmic computation.

Computation in mathematics

The mathematical study of computation emerged from the formalist movement in mathematics. In the 19th century and in the early 20th century, people realized that mathematical theories, used to describe and understand structure, could also be understood as structures themselves. This inspired Hilbert's movement of formalism at the beginning of the 20th century

In the last section of my chapter about the development of understanding in mathematics I already briefly discussed the formalist movement. The formalists argued that mathematical theories had no meaning outside their own theoretical structure. Therefore, the mathematical truths derived from these theories had meaning within that theoretical structure only.

The truth or falsehood or a mathematical statement followed from its consistency with the structure of the theory and nothing else.

Therefore, the formalists understood mathematical truths as being produced by a formal system. Mathematical theories were formal axiom systems and the derivation of mathematical truths was a number of operations on that system, according to a fixed set of rules.

The formalist movement sought to unify existing mathematics in such a formal axiom system. In formalism truth of a statement follows from consistency with the system it is derived from. Therefore, in order to be able to derive mathematical truths, this system has to be consistent itself. This consistency would provide the science of mathematics with the justification the formalists were looking for. This meant that the formalists also needed a proof that this axiom system was consistent. The proof, of course, also needed to consists of formalist 'rule manipulations' (Zach, 2016).

In order to do this, the existing practice of mathematics needed to be described in terms of an abstract, formal structure. Mathematicians started to study the structure of mathematical theories and the structure of mathematical reasoning. From the efforts to describe the structure of mathematical reasoning, the mathematical study of computation emerged.

The first mathematical descriptions of computation

The formalization of mathematical reasoning involved a formal definition of the notion of an *effective procedure* or *algorithm*. As procedures for solving mathematical problems, these algorithms allow us to calculate the output values for a certain mathematical function.

They consist of a clear series of steps, which transform our initial input value(s) into the desired output value. These operations are very important, because they form the link between the inputs of a function and its outputs. Without this clear link, it is impossible to calculate an output from an input. Not all functions have an algorithm that provides such a clear link. And not all clear

algorithms are executable[7]. Therefore, not all mathematical functions are computable.

Providing a formal description of the structure of such algorithms was certainly not an easy job. The formalists sought to understand the different theories of mathematics as belonging to one unified logical structure. Because they wanted to reason about mathematics as one single thing, they needed to describe the structure of mathematics in very general terms. This included algorithmic reasoning.

The formalists were not interested in to describing the structure of specific algorithms. They wanted to describe the mathematical algorithm as a generalized notion. This algorithm would be described in terms of the relationships between its operational steps and the output it provided. Such a generalized description would enable them to understand how in general, mathematical algorithms can provide the desired outputs for functions, without needing to go into the details of specific theories.

Independently of eachother, Alonzo Church (1936) and Alan Turing (1936) developed mathematical descriptions of this underlying structure. By describing this structure, they were able to demonstrate how algorithms work on an abstract level. Their descriptions showed that all algorithms consist of a similar kind of operational steps. For all algorithms, following these steps will lead to correct outcomes in a similar fashion.

Their descriptions were very different from each other. Turing described algorithmic computing with a fictional calculating machine that performed these abstract steps mechanically. His machine could scan and recognize a limited number of input symbols. After recognizing a specific symbol, it mechanically performed an operation that corresponded to the symbol it had scanned. Turing showed how the basic operational steps of his machine allowed it to

[7] It is possible for an algorithm to be clear but not executable. Computer science has shown how this can be the case by describing algorithms that involve consulting a non-existing oracle.

execute all possible algorithms. By doing this, he proved that all executable algorithms consist of the same basic steps.

Church described the structure of computing with a cleverly devised calculating system. His *lambda calculus* consisted of a number of symbols to encode functions. His calculus also had a few simple rewriting rules. The rewriting process of his calculus formed a precise description for the abstract process of computation. All calculation processes in mathematics follow the same pattern as this process. This process allowed him to demonstrate how the steps of algorithmic computation result in the desired output, independent of the specific algorithm that is being carried out.

These descriptions may look very different, but 'under the hood' they are very much the same. They describe the same class of functions. Also, the steps these theories describe produce correct results in a similar fashion. This means that these different descriptions are mathematically equivalent to each other. They use different terms, but they describe the same underlying structure.

These mathematical theories describe all the different algorithms in mathematics with just a few simple rules. These basic steps form the building blocks of all algorithmic computation. With these basic building blocks, you can construct any algorithm you can think of. Therefore, Church's and Turing's descriptions form powerful languages for expressing algorithms. These languages enabled mathematicians and logicians to understand computing and to reason about it.

A few years later, the first stored-program computers were developed. These computers built on the technology used in older types of computer, but also made use of the new mathematical insights about computation.

A mathematical understanding of algorithmic computation

The mathematical descriptions of Church and Turing express our understanding of the underlying structure of algorithmic computation. Although they use different models to describe this structure, computer scientists understand this structure in exactly the same way as mathematicians do.

If computer science would have a different conception of this underlying structure, they would be studying a different kind of structure than mathematics. Then, the term 'algorithmic computation' would pertain to different phenomena in these fields. This is not the case. Computer science studies the phenomenon of algorithmic computation as it was first described by mathematicians in the 1930's.

In mathematics and computer science, the word 'computing' has a specific meaning. To compute something means: to determine the output value for a specific input value of a function by following the steps of an algorithm.

The theories of Church and Turing describe the process of such computation. Computation is a form of mathematical reasoning, especially when it is done by a human: "To calculate the side opposite to the right side of the triangle, I must take the squares of the length of the base and the right side and…" This may create the impression that the theories that describe computation also describe the contents of a mathematical reasoning process. This is not the case, however.

Theories of computation only describe the structure that is underlying these computational processes. They describe the basic steps that are taken in these processes and show how these basic steps relate to the outcomes of these processes. But they do *not* describe the mathematical reasoning behind such algorithms.

They do not tell us how the steps in an algorithm reflect our understanding of the corresponding mathematical function, or why the steps in these procedures will produce correct output values of this function. They do not tell us what the outcomes of these processes actually mean. The algorithmic languages of Church and Turing do not discern total nonsense from correct formulations of established mathematical understanding[8].

These descriptions do not go into the details of these mathematical inferences, because they are created to describe algorithmic

[8] Or, as computer scientists and software engineers often say: 'garbage in, garbage out'.

processes in general. The underlying structure of the computational process is something that all mathematical algorithms have in common. But the mathematical reasoning in these different algorithms is specific to a particular theory.

How programming languages help us to understand algorithmic computation

Before the emergence of computer science, the study of algorithmic computation was a specialized subfield of mathematics. But when people started to think about algorithmic computation in terms of computer programs and programming languages, this subject developed into the blossoming, independent discipline of computer science. Apparently, these programming languages were very fruitful for the understanding of algorithmic computation.

In the next section, I will explain why these languages offer such a fruitful description of algorithmic computation. I believe that these languages are so useful because they are an abstraction of the modern, stored-program computer. As I will show, the abstraction of stored-program computing is a very powerful model for thinking about algorithmic computation.

Programming languages as abstract models of the stored-program computer

Modern programming languages are designed to be abstract. But they are abstracted descriptions of a specific *kind* of algorithmic computation. They describe computation as it occurs in modern, stored-program computers. As I have argued in my previous chapter, these languages do not refer to the details of specific machines. But they *do* refer to a generalized idea of electric stored-program computing. These languages are full of concepts that refer to the way actual stored-program computers function.

For instance, in these languages algorithmic procedures are expressed as *computer programs*, which consist of a set of *input data* and a *sequence of instructions*. This is exactly the way in which a stored program computer receives its data and instructions. These instructions involve the *manipulation* of this input data in a sequential, stepwise fashion. These instructions include operations on the data itself, *reading* and *writing* to *memory*, *memory allocation*

and *jump instructions* that change the sequential order of the program.

This does not mean that these descriptions are fundamentally different from mathematical descriptions of algorithmic computation. As I already explained, computer scientists understand the structure of algorithmic computation in exactly the same way as mathematicians do. All modern, general-purpose[9] programming languages are equivalent to the classical descriptions of Church and Turing.

But although all these theories and languages describe the same thing, they describe this thing in different ways. Church describes algorithmic computation as a mathematical calculus. Turing presents a thought experiment with a hypothetical computing device. And modern programming languages use the abstracted notion of the stored-program computer to describe algorithmic computation. By invoking different models, these descriptions build forth on different bodies of existing understanding.

I believe that programming languages are fruitful because they use the model of the abstract stored-program computer. As I will show in the next paragraphs, this model builds on a special body of existing understanding.

Why this model is so powerful

Programming languages built on our existing understanding of the *actual stored-program computer*. The fruitfulness of these languages shows that this body of understanding proved to be very useful.

I believe this device was able to provide us with such understanding, because it actually is two things at once. As a physical device, it allows us to develop a refined, hands-on

[9] General programming languages are Turing complete. But for specialized programming languages, there are some notable exceptions. For instance, *Vertex Shaders* are very simple processor cores, used for rendering computer graphics. Graphical processor units consist of hundreds or even thousands of such cores, all running in parallel. Older shader models, do not possess a looping capability, which means that they are not Turing complete. Therefore, the specialized languages that were used to write programs for these shaders are also not Turing-complete. See also https://stackoverflow.com/questions/24569439/are-gpu-shaders-turing-complete

understanding while working with it. But at the same time, this device is also a precise description of algorithmic computation.

Modern, stored-program computers can perform any computation from a few basic, hard-wired building blocks. They are built to recognize a particular encoding of algorithms, expressed in a few basic symbols. The standard symbol set for this became the binary symbol set, consisting of nothing more than the symbols 0 and 1. After scanning their input, these machines automatically perform the corresponding operation.

Like the descriptions from Church and Turing, these machines had a symbol set an encoding system and a set of hard-wired 'rewriting rules'. Like the descriptions of Church and Turing, they could express every possible computation with this combination of symbols and rules. And because the rewriting rules and the encoding system had to steer a mechanical device with no human ingenuity, they were expressed in a very precise manner, avoiding any ambiguity or vagueness.

Therefore, these mechanical computers can be viewed as formal, precise descriptions of algorithmic too. These descriptions are mathematically equivalent to the work of Church and Turing. This is because they express exactly the same class of functions: all functions for which a clear and executable algorithm exists. The difference is that these computers describe computation in hardware, instead of describing it in a research paper.

Of course, this was not the only difference here. The models of Church and Turing were designed to describe algorithmic computation as simply and elegantly as possible. But the stored-program computer is a device that actually has to work when you switch it on. To be able to run a program, the computer's processor, instruction set, power supply, input tape, vacuum tubes and many other things interact in a dazzlingly complicated manner. Even the earliest, most basic computers were already incredibly complex. This makes it nearly impossible to comprehend the functioning of such a device all at once.

This enormous complexity seems to disqualify the computer from being a good model for understanding computation. But despite its huge complexity, people were quickly able to develop a substantial understanding of it. This is because the computer is not only a theoretical model, but also a practical machine people have to work with.

Trying to comprehend a thing by looking at it from a distance is not the most effective way to develop an understanding of it. Actually working with a model, interacting with it, receiving feedback from it, can build a much deeper form of understanding. As a flexible device with a practical use, the stored-program computer generously allowed for such interaction.

By learning how to program these computers, the first computer pioneers gradually started to grasp how everything fit together. Through interacting with an actual computer, observing its functioning, these people were able to develop a deep understanding of it.

Because each stored-program computer is a model of algorithmic computation, these pioneers also developed a deep understanding of algorithmic computation in general. They learned to understand how this form of computation worked, because they learned how their stored-program computer did it. As a model *and* a machine, these computers were able to provide a powerful understanding of algorithmic computation.

However, although this body of understanding was powerful, it was far from perfect. Firstly, this understanding was needlessly complex. You do not need to understand the detailed inner workings of an actual stored-program computer to understand how stored-program computing works in general. Secondly, different computers handle the details of computation in a slightly different way. This means that different computers ‘describe’ stored-program computation in a different way. Therefore, the development of the modern stored-program computer did not just offer one model for thinking about computation. It offered dozens of different models.

Therefore, as a model of computation, the actual stored-program computer had some severe drawbacks. To really be of use, this model had to be transformed. It had to be changed into a model that ignored the unnecessary machine details but preserved the general idea of stored-program computation.

In the second half of the 1950's, partially out of the accumulation of practical experience, the notion of the modern programming language emerged (Nofre et al., 2014). These modern programming languages provided such a model.

By using the notion of a stored-program computer, these models built forth on people's existing understanding of stored-program computation. By using an abstract model of stored-program computation, they disconnected specific machine details from the general, underlying idea. This resulted in a much simpler description which facilitated reasoning about this existing understanding. This, in turn, enabled the development of a shared understanding of stored-program computation.

Stored-program computation is a form of algorithmic computation. Therefore, these languages were also fruitful for understanding algorithmic computation in general. Because of this, these languages proved to be very powerful models for the study of computation.

Computer science and empirical science: How the physical can help to understand the abstract

In this chapter, I will relate computer science to the development of understanding in empirical science. In the last section, I explained that computer science seeks an understanding of structure. This means that computer science is not a classical empirical science such as physics.

In the chapters about physics and mathematics, I have shown that these two forms of understanding are entirely different from each other. Therefore, computer science as the study of structure has little to do with the way natural sciences develop understanding.

However, I have also shown that experiences with actual computers were indispensable to the development of computer science. This

'empirical-ness' seems to set the field apart from the field of mathematics also. So, what is it, then? Does it stand somewhere 'in between' those fields? And what would that mean? How exactly, can a scientific field stand 'in between' these different kinds of understanding?

In the last part of this chapter, I will address these questions. I will show how computer science stands 'in between' empirical science and mathematics. I will do this by explaining how the field incorporates empirical elements into the study of abstract structure.

In my chapter about understanding in physics, I discussed how the field of physics develops understanding. The field uses experiments with physical phenomena to infer the nature of the hidden causes behind them. But there are more ways in which experience with the physical can develop our understanding.

In the next paragraph, I will discuss the other way in which the physical brings about understanding. I will show how computer science uses of this way of understanding. Then, I will show how this sets the field apart from the more classical practices of mathematics.

Physical objects as models for understanding

Physical objects can also help us to develop understanding by serving as physical models. When we want to develop an understanding of something, we can create a physical model of this thing. This model represents our understanding of this thing.

Inscribing our understanding into a physical model is useful, because it allows us to 'offload' part of our understanding. By placing our internal understanding into an external model, we free up mental resources. We no longer have to keep the entire thing 'at the top of our heads' when we are reasoning about it. We can choose to play with the physical model of this understanding instead. We can look at the model from different angles and rearrange bits and pieces, to see what happens.

Such 'tinkering' creates a fertile feedback loop between the ideas in our minds and the understanding embodied in the physical model. For instance, James Watson and Francis Crick arrived at the structure of the DNA molecule by playing with cardboard models of the different compound elements of DNA ("The Discovery of the Double Helix, 1951-1953,").

Physical models can help us to develop an understanding of physical things. But they can also help to develop understanding of abstract concepts. For instance, mathematical objects can be represented by knitted and crocheted objects, which often are quite beautiful[10].

Physical processes as physical models for understanding

Physical objects can bring about understanding by serving as models. But we are not only interested in understanding how things *are*. We also want to understand how they *develop*. That means that we are interested in understanding processes too. Processes can also be modeled by the physical.

Physical *objects* represent our understanding of a thing at a specific moment in time, or a specific state. Physical *processes* represent our understanding of how processes unfold over time. The unfolding of the physical model-process reflects the unfolding of the process we seek to understand.

For instance, the behavior of flows and currents in rivers can be studied by mimicking those flows and currents with an electrical current in an analog computer. Before the large-scale introduction of stored-program computers, this is how people calculated the effects of dams, bridges and levees. By using various transistors, resistors and capacitors, people were able to create a flow of electric current that corresponded with the flow of the real river[11].

---

[10] For some nice examples of such knitted objects, see: http://mentalfloss.com/article/86016/6-math-concepts-explained-knitting-and-crochet

[11] This is where the 'analog' in the term 'analog computer' comes from. The electrical current was studied as an analogy for the current in the actual river. The word 'computer' in the term 'analog computer' suggests a close relationship to other kinds of computers, like the digital stored-program computer. And in many respects, these machines are much alike. Both are intricate devices, built to perform complex calculations.

Like ship model basins or wind tunnels, analog computers contain a small-scale physical process, which serves as a model for another process. One example of such an analog model is the MONIAC. This device uses colored water, contained in a series of transparent reservoirs, to model the flow of money in the British economy[12]. It is a beautiful example of how a physical process can serve as a model for our understanding.

The examples of ship model basins and wind tunnels may raise some questions here. On the one hand, these environments can be viewed as models. They are a scale model for the aerodynamic and hydrodynamic processes in our actual skies and seas. But on the other hand, the processes in these environments are not an 'analogy' to the processes we seek to understand. These environments allow us to study the *actual* behavior of water and wind. They are the same processes, only on a smaller scale. Doesn't that also qualify them as actual experiments?

I admit that it is very hard to make a clear distinction between a process serving as a model and an actual experiment in a controlled environment. However, I believe that it is not really necessary to make such a strict distinction here.

In my view, the terms 'model' and 'experiment' are not an either-or category. They pertain to the functions a process can fulfill in the development of understanding. Many processes, like the processes in wind tunnels and water basins, actually fulfill both functions. They bring about understanding by being an experimental phenomenon *and* by serving as a model process.

But from the perspective of modern theoretical computer science, there is a world of difference here. For theoretical computer scientists, the term 'computation' pertains to algorithmic computation. And unlike digital computers, analog computers do not arrive at their results by this kind of computation. They do not carry out a stepwise calculation procedure. They predict the behavior of the process by actually recreating the process (or some aspect of it) on a smaller scale.
Therefore, to the theoretical computer scientist, analog computers are more akin to scale models, than they are to digital computers.

[12] For instance, see https://www.inc.com/magazine/19950915/2624.html The Wikipedia article: https://en.wikipedia.org/wiki/MONIAC

A physical process as a model of computation

I have explained that experience with physical objects and processes can bring about understanding, because these physical things can serve as models. In this section, I will show that computer science makes use of such physical models too.

Computer science, like mathematics, seeks to develop an understanding of structure. The study of structure is the study of the abstract. It is not about understanding the actual world. It is about understanding an abstract concept of this world. People who seek to understand structure seek to develop an understanding of our intuitive concept of structure, as we perceive it in this world.

Structure, however, is not only perceived in the world around us. Certain aspects of mathematical reasoning *itself* can also be understood as mathematical structures. Church (1936) and Turing (1936) showed that the stepwise process of mathematical calculation, could be described as a mathematical structure also.

As a stepwise calculation procedure, algorithmic computation is not only a structure, but also a process. This means that it can be understood by modeling it with a physical process. This is what happens in the insides of a digital computer. The bit-flipping electronic processes in digital computers represent the process of algorithmic computation.

Therefore, the processes in the digital computer provided a physical model for understanding algorithmic computation. The stored-program computer proved to be the most fruitful model. Over the years, people have been working on this type of computer, further developing their understanding of this model.

People found ways to instruct the stored-program computer in human language. They noticed that even the behavior of the simplest program is often very hard to predict. They have learned how to connect different programs seamlessly together, in one 'system program'. They developed ways to let such a system program control different computing processes, in an 'intelligent' manner. By writing bigger and bigger programs, they learned how complex computations can be built from simpler ones. They have developed

ways to manage the inherent complexity of such programs. They learned how to make the most efficient use of the computer's processor and memory. And they learned how input data could best be structured to make this possible.

By working with the stored-program computer, people were able to develop a deep understanding of stored-program computation. And because stored-program computation is a physical model for algorithmic computation, people also developed a deep understanding of algorithmic computation. This allowed the study of algorithmic computation to develop into an independent, blossoming scientific discipline.

Computer science is an abstract field, which seeks an abstract understanding of structure. Therefore, this field does not literarily refer to these experiences with stored-program computers. In computer science, this understanding is described in abstract terms only. But although these concepts are abstract, they are built on experiences with actual stored-program computers.

These abstract insights are applied to the design and operation of actual computers. This changes the practice of working with actual computers. This in turn, leads to new experiences with stored-program computation. This creates new 'practical' understanding, which is a basis for developing new abstract insights. In computer science, abstract understanding and actual experience provide a fertile feedback loop, enabling the development of new understanding.

Or, as Edsger Dijkstra (1989) put it, "computing science is -and will always be- concerned with the interplay between mechanized and human symbol manipulation" (p. 5).

Mathematical reasoning as a physical process

In the previous section I explained that we can place part of our understanding outside us, by creating physical models. These models are external representations of our understanding. I have shown that computer science also externalizes its understanding in a physical model. By externalizing its understanding, computer science adds an interesting twist to this story.

Algorithmic computation is an aspect of our mathematical reasoning process. Therefore, by creating a physical model of this process, we create a physical model of our own reasoning process. This means that we are placing this reasoning process outside us. Our mental reasoning process becomes a physical phenomenon, which can be observed.

In this respect, the newspaper headlines of the 1950's about 'machines that think', were spot-on. Algorithmic reasoning has become something the machine can do for us. We press the buttons of the machine, to establish the outcome of an algorithmic reasoning process.

How a physical process creates an 'empirical' mathematics

This 'twist' sets the field of computer science apart from many other fields in mathematics. It changes computer science into a partially empirical field. While mathematics studies our concept of structure by reasoning, computer science studies the structure of this reasoning itself. This mental reasoning process is also an abstract structure. But because it is externalized into a physical process, it becomes a physical phenomenon. This enabled computer science to become empirical.

As a physical phenomenon, algorithmic computation allows for interaction. It can be tested and observed. The possibility to test and observe is vital to understanding algorithmic computation. This is because algorithmic processes are very complex. Therefore, even the outcomes of the most simple of programs are often impossible to predict beforehand.

This interaction creates a powerful feedback loop between people's ideas and the physical phenomenon. Through their interaction with this phenomenon, computer scientists were able to develop a deep understanding of algorithmic computation.

Because of this, computer science stands in between empirical science and mathematics. Like mathematics, it seeks an understanding of abstract structure. But the structure it studies has a specific nature. This allows it to be externalized into a physical process.

By studying this physical process, computer science is able to study this structure in an empirical manner.

References

Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics, 58*(2), 345-363.

Dijkstra, E. W. (1989). On the cruelty of really teaching computing science. *Communications of the ACM, 32*(12), 1398-1404.

The Discovery of the Double Helix, 1951-1953. *The Francis Crick Papers.* Retrieved August 21, 2017, from https://profiles.nlm.nih.gov/SC/Views/Exhibit/narrative/doublehelix.html

Nofre, D., Priestley, M., & Alberts, G. (2014). When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960. *Technology and culture, 55*(1), 40-75.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math, 58*, 345-363.

Zach, R. (2016, Spring 2016). Hilbert's program. *The Stanford Encyclopedia of Philosophy.* Retrieved August 19, 2017, from https://plato.stanford.edu/archives/spr2016/entries/hilbert-program/

Chapter 8: Computer science as a science

Introduction

In this thesis I wanted to develop a better understanding of computer science as scientific field, by showing how computer science was able to develop new scientific understanding. I have formulated this aim as a research question with two sub-questions.

Research question: How can the field of computer science develop new understanding?

Sub-question 1: What kind of thing does computer science seek to understand?

Sub-question 2: How does computer science develop an understanding of this thing?

In the previous chapters, I have related computer science to physics and mathematics. The development of understanding in physics and mathematics better corresponds to our classical ideas of doing science. By linking computer science to the understanding in these fields, I was able to explain the understanding developed in computer science in familiar terms.

In this concluding chapter, I will use the results of this discussion to answer my research question. I will reflect on the strengths and weaknesses of my approach. Then, I will discuss the implications of my results and give some recommendations for future research.

Sub-question 1: What kind of thing does computer science seek to understand?

In my thesis I have been able to answer the first sub-question. I have shown that computer science seeks to understand algorithmic computation as an abstract concept.

Algorithmic computation is the process of calculating an output from the input of a specific function by following the steps of a mathematical solution procedure. Humans can perform algorithmic computation. But the steps of an algorithm can be carried out by a machine too.

Computer science seeks to understand these algorithms as an aspect of our intuitive concept of structure. In this sense, computer science is akin to mathematics. In my chapter about mathematics, I have shown that mathematics is able to develop understanding through developing our intuitive concept of structure.

We are born with an intuitive idea of the structure of reality. By interpreting reality in terms of this structure, we are able to understand it. We see structural patterns in the winds and waves, in the size of animal populations, in the behavior of human beings and everywhere else. The field of mathematics seeks to further develop this concept of structure.

We understand algorithmic computation as a structure too. This structure was first described by Church (1936) and Turing (1936). Computer science has refined our understanding of this concept of structure. This means that computer science does not actually seek to understand actual algorithmic computation processes. It seeks to develop our understanding of the underlying structure of these processes. Computer science therefore, seeks to develop an understanding of algorithmic computation as an abstract concept.

Sub-question 2: How do they develop an understanding of this thing?

In my thesis, I have also been able to answer my second sub-question. I have found out how computer science is able to develop an understanding of the structure of algorithmic computation. Computer science combines reasoning about formal descriptions of algorithmic computation with experience with a physical model of algorithmic computation.

In my chapter about mathematics, I explained how the different subfields of mathematics seek an understanding of the different aspects of our intuitive concept of structure. When mathematicians want to further develop their understanding of a specific structure, they try to precisely articulate their intuitive ideas about this structure. This enables them to create a precise description of this structure. This description provides a clear starting point for further inference about this basic structure. By reasoning about

this basic structure, they learn what this basic structure entails and what kinds of things follow from it.

Computer science took quite a different route. The study of algorithmic computation started like many other mathematical subjects. Mathematicians articulated their intuitive ideas about the structure of algorithmic computation into a precise, formalized description (Church, 1936; Turing, 1936). They used this structure to make mathematical inferences. But after a few years, something else happened: the first stored-program computers were developed.

Because stored-program computers are physically able to perform the process of algorithmic computation, they are a physical model for this process. By externalizing their understanding of algorithmic computation in a physical device, people had created a machine that was able to do such computations.

Mathematical descriptions of structure bring understanding because they allow you to reason about them. But this 'description' could do parts of this reasoning itself. People could observe what their concept of algorithmic computation entailed, by putting this concept to work. This model afforded a kind of interaction that traditional mathematical descriptions could not provide.

Actually working *with* a model, interacting with the model and receiving feedback from it, can build a deep form of understanding. This understanding may be much deeper that the understanding you develop by reasoning *about* a model. By working with the first stored-program computers, many people were able to develop a deep understanding of algorithmic computation, whether they realized this or not.

In the late 1950's and early 1960's, the modern conception of a programming language emerged. Programming languages are formal, mathematical descriptions of the structure of algorithmic computation. These models described algorithmic computation with an abstract concept of the stored-program computer. By offering an abstract description of the stored-program computer, this model built forth on people's existing understanding of stored-program computers. People were now able to express and communicate the deep

understanding of computation they developed by working with actual computers.

This newly developed understanding gave the mathematical study of algorithmic computation a new impulse. People used their existing understanding of stored-program computation to study algorithmic computation in an abstract sense. This new approach proved very fruitful. It enabled the mathematical study of computation to develop into the new field of computer science.

These new computer scientists had taken an interesting detour in their path towards understanding the structure of algorithmic computation. Usually, people begin with formalizing their intuitions in a mathematical model. Then they use that model as a starting point to develop their understanding further. But in computer science, many people started with developing their understanding by working with a model (the computer) and then developed formalized descriptions of their understanding.

Research question: How can the field of computer science develop new understanding?

I have shown that computer science seeks to understand the structure of algorithmic computation. This means that computer science is a study of the abstract. But, I have also shown that practical experiences with actual computers were indispensable for the development of this understanding.

Answering my two sub-questions brings me to my final research question. How can the field of computer science develop new understanding?

In my thesis, I have shown that this field emerged out of experiences with actual computers. These practical experiences led to abstract insights about the structure of algorithmic computation. The abstract insights were applied to the design and operation of actual computers. This influenced the practice of working with actual computers. This created new 'practical' understanding. This new understanding, in turn, was a basis for developing new abstract insights, which influenced the development of actual computers. And so on. The emergence of this field had started a fertile feedback

loop between the physical and the abstract. Through this feedback loop, computer science can develop new understanding of the structure of algorithmic computation.

This practice sounds similar to the development of understanding in applied science and engineering. For instance, in the field of aerodynamics, practical experience with actual aircraft generates understanding of airflow behavior. This leads to the development of mathematical models that describe this behavior. These mathematical models influence the development and design of new aircraft, which provide new practical experience, which leads to new models, and so on.

But there is a difference between these two practices of developing understanding. The field of aerodynamics is using abstract mathematics to develop an understanding of a physical process. Mathematical models of airflow behavior are abstract descriptions of structure, which are used to describe the behavior of actual airflow. By going back and forth between the mathematical model of the process and the actual process, the field is able to refine its model and thereby their understanding of the physical process.

The field of computer science is doing the reverse. It makes use of a physical process to develop an understanding of an abstract, 'mathematical' structure. By going back and forth between the physical process and the mathematical descriptions of the abstract structure, they are able to refine their understanding of this abstract structure.

The strengths and weaknesses of my analysis

In my thesis, I have chosen to adopt a specific approach. I have chosen this approach because I believed that this approach was best able to answer my research questions. However, this approach also has some weaknesses. In the next section I will reflect on the strengths and weaknesses of this approach.

In my thesis, I explained understanding in computer science by relating it to other forms of scientific understanding. Because of this approach, I could use familiar terms to explain understanding in computer science.

But there are some drawbacks to this approach also. By analyzing understanding in computer science ‘in familiar terms' I have analyzed understanding in computer science in terms of what I already know. Therefore, I might have missed some important aspects of this new kind of understanding, which do not really fit these traditional forms of scientific understanding.

Also, my approach requires me to have a clear conceptual description of those other forms of understanding. In order to be a clear conceptual description, those concepts have to abstract away from the many nuances and complications of the actual scientific process. Therefore, my description ignores many of such nuances and complications.

I did not study real lab rooms and faculties, to uncover the process of developing understanding as it actually happens within the rich context of the scientific enterprise. I also did not go into issues such as quantum mechanics, or the ‘mathematization’ of fundamental physics.

Developments in quantum mechanics have challenged the assumption that everything must have a cause, which is an important part of the metaphysical worldview underlying physics. And currently, in the most fundamental branches of physics, research practices consist for a great deal of developing mathematical models to describe the fundamental properties of matter. In my chapter about physics, I did not discuss what those issues meant for the development of understanding in physics. Therefore, my descriptions of the development of understanding in physics and mathematics do not reflect the actual development of understanding in those fields.

My description of understanding in computer science is based on a historical study of computer science. As I have argued in my chapter about methodology, there are good reasons to use a history study for such a description. Also, there are good reasons to believe that this history study also reflects the development of understanding in computer science today. That is why I have chosen this approach. But there are also drawbacks to this approach. My description does not

reflect the full complexity of all the current practices in computer science.

A better understanding of computer science

Because of these weaknesses, my analysis may not capture the full reality of the development of understanding in computer science. However, as I explained in my introduction, the goal of my thesis was never to show to what extent computer science is *really* like mathematics or physics. My intention was to find a way to understand and talk about the science part in computer science. I believe that I have succeeded in fulfilling this ambition.

My analysis provided me with a set of concepts that allowed me to talk about understanding in computer science in a clear manner. These concepts have allowed me to tackle the confusion surrounding computer science, by 'demystifying' the development of understanding in computer science.

To many people, it is not always clear how the field is able to combine elements from two very different kinds of inquiry. Therefore, many people have come to view computer science as a mysterious mix of mathematics and natural, empirical science. Newell and Simon (1976) have beautifully formulated this confusion:

We would have called it an experimental science, but like astronomy, economics and geology, some of its unique forms of observation and experience do not fit a narrow stereotype of the experimental method. None the less, they are experiments. Each new machine that is built is an experiment. (p. 114)

My descriptions of mathematics and physics gave me a clear conceptual picture of the development of understanding in these fields. This allowed me to explain the development of understanding in computer science. I was able to identify the different mathematical and empirical elements in this development process and to show how they worked together to produce new understanding. I demystified the development of understanding in computer science, by showing that it seeks an understanding of structure, through using a physical model.

It could very well be possible that not everyone will agree with my conclusions. But even then, I believe that I have succeeded in providing a better understanding of computer science as a science. This is because my analysis has provided a set of clear concepts and relations. These enabled me to think and talk about the practices of computer science. And, as Boon (2009) has shown, clear concepts and relations are the key to developing new understanding. Hopefully, this analysis will be helpful to others in formulating their own views.

Implications for future research

My findings point to a number of interesting avenues for further research, which may help us to develop a better understanding of computer science as a science. In this final section of my thesis, I will discuss some of these issues.

Analyzing a new way of doing science

My thesis shows that computer science has created a new way to understand structure. Therefore, the practices of computer science could be viewed as a new way of doing science. What kind of issues did computer scientists encounter when pioneering with this new way of doing science? How do these issues relate to the nature of their 'model'? And how did they find ways to deal with those issues?

Also, this new method of doing science is hard to grasp for many people. Many even contest that computer science is a science. Somehow, we do not clearly see what computer science *is* and *does*. The methodology of other sciences, such as biology and physics, seems much more straightforward.

You do not need to understand Darwin's theory of natural selection, or the Standard Model of Particle Physics itself, to see what those theories *are for*. Darwin's theory of natural selection explains why the organisms in the world are the way they are. And the Standard Model explains why the fundamental bits of matter in this universe behave the way they do.

But when it comes to computer science, the uninitiated find it much harder to grasp what theoretical models such as the as the Church Lambda Calculus are meant to tell us. And the fact that the

corresponding explanatory texts (written by the initiated!) repeatedly stress that the model is 'very simple and elegant' does not make it much easier.

Is there a connection between the confusion about the methodology of computer science and the difficulties many novices experience in grasping concepts of the field? Is it because this field is relatively new? And if the methodology of computer science is not so well understood as the classical 'scientific method', would it lead to more reflective practices if this methodology would be explicitly formulated in a set of 'basic principles'?

Practical experience and the black-boxing of a physical model

The black-boxing of the computer, which I described in my thesis, points to some interesting issues. I argued that practical experiences with the first stored-program computers were crucial to developing an initial understanding of algorithmic computation. When people started to reason about computation in an abstract sense, their understanding of computation was still very close to the inner workings of the actual machine. The understanding of abstract computation they developed was based on this practical understanding.

But when modern programming languages were introduced, people no longer needed to know every processor detail to operate a computer. The tablet-swiping toddler and the fact that people nowadays can develop apps without understanding what a compiler is, testify to this. People's 'practical' experience with the actual computer became increasingly abstract itself. How did this influence the development of understanding in computer science?

And how did this change our understanding of computer programming? What does programming in a modern-day language teach us about algorithmic computation? And how can understanding of algorithmic computation help us to understand modern-day programming? Is there a relationship between the black-boxing of the computer and specific errors made by inexperienced and unreflective programmers?

A computational understanding of nature

Next to these questions about practice and methodology, the results of my thesis also point to another interesting issue.

In my thesis, I have shown that computer science seeks to develop an abstract understanding of structure. But there are many computer scientists who are convinced that computer science is not confined to the study of the abstract. For instance, Denning (2007); Eden (2007); Newell and Simon (1976) strongly believe that computer science is studying aspects of the natural world. With the findings from my thesis, I can explain why many people feel that computer science is a natural science. These views provide some interesting avenues for further research, as I will show.

In my thesis, I explain how computer science was able to develop a new way of understanding abstract structure. By creating the first stored-program computers, people externalized their understanding of algorithmic reasoning in a physical process. This new connection enabled the field to develop a new understanding of structure.

But this connection works in the opposite direction also. If people learn to understand *computation as physical processes* they also learn to understand *physical processes as computation*. Therefore, next to helping us to understand the abstract, computer science also offers a new way to understand the physical.

This new way of understanding physical processes is not at all like the 'classical' understanding of the physical, described in my chapter about physics. In the classical way of understanding, you understand physical processes as chains of cause and effect. You would explain the flip of a bit inside the memory in a computer as caused by an electrical charge crossing a certain threshold, after which the hidden forces of electromagnetism cause the magnetic charge of the bit to reverse. But when you understand the flipping of a bit as computation, you view the flip of a bit in a computer as the alteration of a specific value during the execution of an algorithm.

This new kind of understanding proved to be fruitful in understanding some aspects of the physical world. It helps to

explain processes that are hard to understand in terms of cause and effect only. Think, for instance of a complex organism developing from a DNA molecule in a fertilized egg.

In principle, it could be possible to understand this process in terms of cause and effect. But the causal chain between the chemical properties of the DNA molecule and the characteristics of the adult organism is very intricate. Trying to understand this process in terms of cause and effect would be like trying to understand the chain of events between the proverbial butterfly flapping its wing and the resulting hurricane. There are specialized biologists who understand different parts of this causal chain. But for us humans, it is probably too complex to understand in its entirety.

When we understand this process as a computation, we are much better able to understand this process. We generally conceive of the DNA molecule as an information carrier, instead of a causal trigger. We understand that organisms can develop from the DNA in a fertilized egg because the molecule acts like a computer program, providing the fertilized egg with a set of coded instructions, necessary for the further development of the organism.

Therefore, next to developing a new way to understand structure, computer science also contributed to a new way of understanding physical processes. This new way proved fruitful for understanding complex phenomena in the natural world. This is why some computer scientists view computer science as a natural science.

For instance, Eden (2007) argues that computer science is the study of ‘computational program-processes’. He views these program-processes as naturally ocurring phenomena in our phyiscal world, on a par with DNA sequences. Denning (2007) points out that insights from computer science have been used to study the informational properties of complex molecules (such as DNA) and that quantum physicists vieuw the behaviour of quantum particles as an information process.

But although computer science helps us to develop an understanding of the physical world, this does not mean that computer science is a natural science. This field seeks an understanding of algortitmic

computation as an abstract concept. Of course, this understanding can be used to understand actual, physical processes too. But this understanding is only the indirect product of computer science. And even if you are willing to stretch the idea of a natural science to include computer science, I have shown that the understanding it develops is not at all like the 'classical' understanding developed in the other natural sciences.

Still, the articles of Denning and Eden do point to an interesting avenue for further research. These authors have realized that the notion of computation offers a new way to understand the physical world. How can we characterize this computational understanding of the world? And to what extent, did developments in computer science contribute to this new form of understanding?

How algorithmic reasoning became a part of the structure we see in the world

Perhaps, further research shows that the development of computer science was somehow crucial to the development of this new understanding. Then, algorithmic computation has made a journey that is interesting for philosophical study. From something that once coincided with our mathematical reasoning, it was placed outside us and became part of the structure we perceive in the physical world.

When mathematicians realized that algorithmic reasoning could be understood in terms of structure, the first distance between us and algorithmic reasoning was created. Algorithmic reasoning no longer coincided with our mathematical reasoning. It became a subject of this reasoning, something we could reflect on and study.

The next step was to externalize our understanding of this subject. First, we externalized it in a classical, mathematical description. Then we modeled this subject in a physical, working machine. Now, the distance between us and algorithmic computation was even bigger. It had become a physical phenomenon, a part of the outside world, something we could observe.

And perhaps, because we had placed algorithmic computation in the outside world, we were able to recognize its structure in other things in this outside world. We started to interpret DNA processes

as instances of this structure. A concept of something that once coincided with us is now used to understand the outside world. From a philosophical viewpoint, it would be interesting to chart the travels of algorithmic computation, as part of our concept of structure.

From mathematical reasoning to physical process

There is one final avenue for future research that I would like to discuss here. In my thesis, I showed that people externalized algorithmic computation in a physical model, the computer. It would be interesting to find out to what extent, this altered people's understanding of the nature of computation.

As I already explained, computer scientists seek to understand algorithmic computation as an abstract concept. They are not interested in studying how a particular program can run on a particular computer.

But the abstract concept they seek to understand is an abstract concept of a physical process, limited in space and time. In computer science, the complexity of algorithms is expressed in the memory space and processor time needed to execute them. An algorithm can be claimed to be impossible to run, because executing it would require more bits than there are particles in the entire universe.

Apparently, the mental, mathematical activity of algorithmic reasoning has become to be viewed as a physical process, with physical limitations. Could it be that our idea of algorithmic computation changed because we externalized it in a physical computer?

The physical, stored-program computer used energy and produced heat. It had a limited amount of space in its memory banks. Also, the time needed by the processor to carry out an operation put a limit on the speed at which programs could be run. Perhaps, the physical limitations of the computer made it apparent that algorithmic computation, as a physical process, was constrained by finite resources in space and time.

Future research could focus on finding out to what extent the externalization of algorithmic computation changed our ideas about the nature of computation. Also, it would be interesting to see how such a change could have influenced our theoretical understanding of algorithmic computation.

References

Boon, M. (2009). Understanding in the Engineering Sciences: Interpretative Structures. In H. W. de Regt, S. Leonelli, & K. Eigner (Eds.), *Scientific Understanding: Philosophical Perspectives*. Pittsburgh: The University of Pittsburg Press.

Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics, 58*(2), 345-363.

Denning, P. J. (2007). Computing is a natural science *Communications of the ACM* (Vol. 50, pp. 13-18).

Eden, A. H. (2007). Three paradigms of computer science. *Minds and Machines, 17*(2), 135-167. doi:10.1007/s11023-007-9060-8

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM, 19*(3), 113-126. doi:10.1145/360018.360022

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math, 58*, 345-363.