Making Construals In The CONSTRUIT Environment and Computing Education

Kenneth Bugembe

Discrete Mathematics - USCA-G4G1 19/20

Supervised by Mike Joy on behalf of Meurig Beynon (Emeritus Reader)

Abstract

There is growing recognition in the United Kingdom that everyone, regardless of their future occupation, will need some capacity to interact with computers as well as knowledge of both Computing and programming.

However, there is increasing concern that current approaches to British Computing education are not adequate for this agenda. That the current curriculum is failing to properly equip students with the knowledge and skills to cope with an increasingly digital world and that the vast majority are choosing not to study Computing past Key Stage 3.

Many argue that it is the difficulty of programming hindering the success and uptake of Computing education. Through the CONSTRUIT! project, the Empirical Modelling Research Group promote the theory of Empirical Modelling as a complementary experiential account of Computing. From this account, the practice of "making construals" arises; a mode of computer interaction they claim is more accessible to non-computing specialists and therefore better suited to an agenda of widely promoting Computing than conventional programming.

This project aims to evaluate the theoretical and practical work of the CONSTRUIT! project in order to determine whether the principles and practice of making construals can be used to improve British Computing education.

Keywords: Making Construals, CONSTRUIT, Programming, Computing Education

Introduction

In the white paper for Computing at School Working Group (2010), they discuss their concern that British students were losing interest in Computing and the group's overall objective of promoting the study and teaching of Computing within schools. The contention of their group, and other educational initiatives promoting Computing in the United Kingdom, is:

> "*The UK needs school-leavers to be familiar with rigorous computing principles if the economy is to be competitive over the long term… although the technology is changing constantly, the principles that future technologies are built on do not,*" (BCS Academy, 2011).

Their concerns were borne out when the Royal Society (2012) in the Executive Summary of their final report on the state of Computing Education in UK schools conclude that in 2012:

> "*The current delivery of Computing education in many UK schools is highly unsatisfactory….many pupils are not inspired by what they are taught and gain nothing beyond basic digital literacy skills...*,".

As a result of such concerns, the Department of Education (DfE) has taken steps to promote Computing Education within British primary and secondary schools. For example, the Computing Programme of Study (DfE, 2013) for the National Curriculum in England aims to allow:

> *"...pupils to use computational thinking...to understand and change the world….pupils are taught the principles of information and computation, how digital systems work and how to put this knowledge to use through programming…pupils are equipped to*

use information technology to create...a range of content...pupils become digitally literate...at a level suitable for the future...".

However, the Royal Society (2017) conducted a subsequent report on the effects of the 201changes introduced by the DfE on British Computing education and state in their Executive Summary:

"Today, 70% of students in England attend schools offering GCSE computer science...However...only a disappointing 11% of all students take GCSE computer science...the range of qualifications on offer does not reflect the full breadth of computing…… The qualification landscape needs urgent attention to ensure the broadest range of pupils become equipped with relevant digital skills,".

In recognition of this, research into Computing education is of increasing vital importance. The Royal Society (2017) notes that:

"*Without this research, the ability and speed with which we can advance our understanding of pedagogies and assessment for computing is limited. Research will provide a strong theoretical base to improve pupils' learning outcome,*".

For Computing education to flourish, research is needed to identify existing gaps in current approaches to British Computing education and to determine how these deficiencies can be addressed.

One line of critique argues that the current conceptualisation of programming is responsible for the deficiencies in Computing education. Granger (2015) discusses the difficulties generated by the fact that programming is not suited to modelling real-life systems: "*We are employing a set of tools (programming) designed to model how computers work, but we're representing systems that are nothing like them.*" In addition, Victor (2012) argues that the difficulty of contemporary programming lies in the fact you are forced to think like a machine and that programming needs to be changed "*...into something that's understandable by people,*".

The general insight to which the above authors allude is that our current conceptualisation of programming (and more broadly computing) hinders our capacity to think with computers. By altering this conceptualisation, there is the potential to increase the capacity for computers to support our thought processes. The implications of this, with regard to education, is that computing should be reconceptualised so that it is both more accessible and generally applicable to students.

One attempt to do so can be seen in the work of the Empirical Modelling (EM) Research Group, which partly explored the notion of "*...making artefacts to support human thinking.*" and how we can use computers to do so (Beynon, 2006). In doing so, the EM Research Group note that there are aspects of programming that the current conceptualisation of computing fails to describe adequately; there is no formal acknowledgement of the experiential aspects of programming. Consequently, they saw their theory of Empirical Modelling as a complementary account of computing focusing on this experiential aspect.

Under an EM framework, software development is characterised by the notion of *"programming as modelling"* (Beynon, 2012b). The programmer needs to understand the scope for agency within his application domain before he is able to begin specifying their program; the programmer's understanding is rooted in "*...the immediacy of the developer's experience of the relationship between software as an artefact on the computer and software as an agency in the world.*" (Beynon, 2012a).

Consequently, the EM approach to improve the ability of computers to support human thinking is to work on developing tools which support the programmer's attempts to understand the application domain. Specifically, this could be achieved by strengthening the connection (as experienced by the programmer) between the software domain and the application domain and supporting a manner of software development that is rooted in observation and experimentation. This work culminated with the concept of "making construals" as a new mode of computer interaction and their subsequent work has focused on promoting this skill in various areas.

A construal is described by Beynon et al. (2015a) as "*...an interactive artefact that we develop to help us explore and record our emerging understanding,*". In the process of making a construal, the components of the desired construal will be constructed and the dependencies between them defined in a way that they correspond to particular features of a referent in the domain of interest. Through repeated experimental interaction and reflection with both construal and referent, the correspondence between the states of the construal and referent would be gradually strengthened, and consequently our understanding of the subject domain.

CONSTRUIT! was a three-year European project, led by the EM Research Group, which examined the skill of making construals, how it could benefit education and created resources which facilitates the creation of digital construals by both educators and learners. The project began as a part of the European Union's Erasmus+ programme with a specific focus on "*...enhancing the quality and relevance of the learning offer in education, training and youth work by developing new and innovative approaches and supporting the dissemination of best practices.*" (European Commission; 2014). The EM Research group perceived this programme as an opportunity to "*...further develop and disseminate the principles of 'making construals' as a computing technology that can have significant impact on how ICT and technology-enhanced learning are regarded and practised.*" (CONSTRUIT! team; 2014).

Their claim was that " *...so much conventional software development is directed towards implementing the dependencies that connect software to its domain of application, and that in principle can serve to link software development and evolution to knowing and learning within the domain,*" (CONSTRUIT! team; 2014). Consequently, they believed that by explicitly strengthening this link between software development and domain learning, making construals was better suited to support education than conventional programming (or ICT). Moreover, they believed that as a mode of computer interaction, making construals is "*...more accessible than conventional programming but more expressive and powerful than conventional uses of ICT.*" (CONSTRUIT! team; 2014).

The CONSTRUIT! team (2014) envisioned teachers and students collaborating in the construction of digital construals with relation to a specific subject domain; these construals would enable "*...educational practices combining instruction and construction.*" Construals

would act as open educational resources (OERs) which could be collaboratively developed and shared amongst peers in order to promote understanding of the subject domain.

In terms of outputs, CONSTRUIT! produced a number of digital construals and a series of publications about the theory and practice making construals. The set of digital construals contains examples of educational resources, tutorials for the Making Construals Environment (MCE), and discussion and illustrations of the concepts behind making construals.

This project aims to evaluate the theoretical and practical work of the CONSTRUIT! project in order to determine whether making construals can improve British Computing education. This will involve examining both the literature produced during the CONSTRUIT! project as well as the Making Construals Environment in its current incarnation.

Regarding the role of making construals within British Computing education, this project will consider two potential proposals:

1. Making digital construals to support Computing Education as currently conceived.
2. Making digital construals as a valuable skill in and of itself that should be taught as part of Computing Education.

Moreover, this project will consider the implications of the work of the CONSTRUIT! project with respect to future trends in Computing education.

Methodology (Theory)

In order to begin the evaluation of the CONSTRUIT! project, I initially chose to focus on determining if a theoretical argument could be made for making construals being utilised in Computing education. Specifically, I sought to answer whether making construals could support traditional Computing education and whether making construals should be taught as part of Computing education. In order to do so, I began by reviewing the publications made by the EM Research group. I felt this was necessary to justify my examination of the current resources for making digital construals (i.e. the MCE), as well as to motivate future work.

To elaborate, I believed it was feasible to conclude the existing resources for making construals were not sufficient to incorporate making construals into Computing education (in contrast to existing educational technology). Moreover, given the general scepticism from the orthodox Computing community (Beynon, 2017b), I also assumed there was a possibility that the technical work of the CONSTRUIT! project was not novel. However, if a convincing theoretical argument could be provided for making construals, this then would justify both this project and further work on making construals, even if I also came to the above conclusions.

I specifically chose to look at publications made during the project's lifetime (i.e. 2015-2017) as, both theoretically and practically, they would have the most relevance to the current incarnation of the MCE (and making digital construals). Moreover, since the work of the CONSTRUIT! project aimed to promote making construals for educational purposes, these papers were most likely to address the issues directly that need to be considered in evaluating the usefulness of making construals for Computing education.

Near the beginning of this project's timeline, I also considered looking at older EM publications specifically concerning educational technology. Since the CONSTRUIT! project (both theoretically and practically) derives directly from these past publications reviewing this literature would give further insight into the work of the CONSTRUIT! project as well as potentially provide a basis for further theoretical arguments. However, due to time constraints and waning relevance (given that this set of publications span back to 1988), I chose as an alternative to examine only older EM papers directly referenced within publications of the CONSTRUIT! project as those would hold the most value to my project.

To ensure the credibility of any theoretical arguments made from the work of the CONSTRUIT! project, I also planned to review relevant literature, outside of Empirical Modelling, concerning Computing education. This was in order to determine to what extent these arguments are supported or opposed by more orthodox work in Computing education. However, the existence of a recent literature review on Introductory Programming by Luxton-Reilly et al. (2018), and one of Computer Science Unplugged pedagogy by Bell T. & Vahrenhold J. (2018) meant I was not required to review this literature for my project.

In terms of the flaws of this approach, it is entirely reliant on secondary data in order to support a theoretical argument for making construals. For EM derived literature, this is not a significant issue given the fact this project's aim is to evaluate the CONSTRUIT! project. However, this does weaken claims that orthodox work in Computing education supports the theoretical arguments for making construals because making construals is not directly discussed within these works. This project mitigates this issue by relying on secondary data outside of EM as a means to highlight issues in Computing education that both EM-based and orthodox work agree exist. EM-based work is then used to argue why making construals could theoretically resolve these issues.

Moreover, this approach also weakens claims concerning how students and teachers would engage with making construals in contrast to conventional programming due to this lack of primary data.

In hindsight, this project should have incorporated an empirical study with directly comparing how students and teachers without a Computing background engage with making construals in comparison to conventional programming. This would have not only strengthened the theoretical arguments in favour of making construals but also give us primary data on whether the currently existing resources for making construals are sufficient to be incorporated into existing Computing education. However, I felt the secondary data was a sufficient basis to make theoretical arguments concerning these claims that would justify future empirical work.

Theoretical Content

There are a number of themes running through the publications of the CONSTRUIT! project which highlight specific gaps in the current approach to Computing Education in British secondary schools that indicate potential roles making construals could serve in British Computing Education.

Construals as an aspect of Computing Education

One potential role for making construals is as a skill taught as a part of the Computing curriculum. As to if there is a case to do this, we can begin by considering what is involved in learning to program.

Beynon et al. (2015a) discuss the notion of the "programmer interface" whereby possessing a degree of specialist Computing knowledge is a prerequisite to learning to program effectively. For teachers without this knowledge, this constrains their ability to tailor programs and software used during Computing courses to their specific needs. This concern is of particular note for teachers in British Primary and Secondary education who are unlikely to have an educational background in Computing given its fairly recent rise in prominence.

The reason this knowledge is needed is due to the gap between the semantics of a program and *"the semantics of the semantics"* (Beynon et al., 2015b) of the said program which programmers must contend with; the effects of a program on the computer's state vs. the effect of the changes of the computer's state on the application domain itself.

To elaborate, the programmer has specific functional objectives they wish to meet within the application domain. In order to do so, the programmer needs to work out what actions must be taken in the application domain to fulfil these objectives. The problem lies in the fact that the programmer has to act indirectly; the programmer is not themselves taking actions in the application domain but rather relying on the computer to act in the application domain. Moreover, the computer holds no concept of the context within which it is carrying out a particular program; the way the computer interprets the meaning (i.e. semantics) of the program is completely independent of the application domain. Consequently, it is up to the programmer to determine how the program they have written will be interpreted by the computer and then determine how the actions taken by the computer will change the state of the application domain. The programmer will fail to meet his objectives if either he does not understand the necessary actions to achieve his objectives in the application domain, or how to instruct the computer to take the required actions to meet his objectives.

This viewpoint aligns with a Constructivist philosophy of education which argues that new knowledge is constructed by learners using their prior knowledge and experiences. Various papers taking a Constructivist perspective have discussed the difficulty this process presents for students. A literature review by Luxton-Reilly et al. (2018) on Introductory Programming explored this topic stating that among the reviewed papers:

"Students were found to hold misconceptions and non-viable mental models of these fundamental concepts even after completing their introductory programming courses," and more concerningly *"...students holding non-viable mental models sometimes still manage to do well on related programming tasks, suggesting that assessment techniques beyond conventional code-writing tasks might be needed to reveal certain misconceptions,"*.

These excerpts not only demonstrate how difficult learning to program is for students but also suggest that non-specialist teachers will struggle to teach programming to students effectively unless they take the time to learn the requisite specialist knowledge.

One immediate response to this issue is to explicitly teach students in viable models of a computer; this was suggested by Ben-Ari (1998) when he examines how a Constructivist pedagogy would work within Computer Science Education. However as discussed by Beynon (2009), if we also accept a Constructivist theory of knowledge then this approach is not sufficient. This is because a student's prior knowledge and experience will influence how well they understand a particular representation of the computer. Consequently while explicitly teaching the model will mean that more students will hold viable mental models of the computer, there will still be a large group of students who will fail to form a viable mental model. Beynon (2009) notes that Ben-Ari's objection to a Constructivist epistemology is built on the formal basis of traditional Computing; he characterises Empirical Modelling as " *an alternative approach to computing that embraces a constructivist epistemology*" which is able to do so because it accounts for the experiential aspects of Computing.

Boyatt, Beynon & Beynon (2014) discuss the need to acknowledge *"programming as a lived experience"* in order to resolve the aforementioned issues. They argue that the contemporary conception of programming as learning to *"...express computational recipes for machines in*

an abstract programming language," ignores the role of experience in the process of mental modelling highlighted above; specifically, they note that programming in practice can be characterised as "*a core activity for the implementation of software applications, where the concrete real-world interpretation of abstract program constructs...must inform the practical construction of programs*,". Consequently, the lived experiences of the programmer are given insufficient treatment with regard to traditional Computing.

The theoretical underpinning of the CONSTRUIT! project begins by acknowledging the importance of lived experience in programming. Accordingly, the computer is conceived of as a source of experience which *"....can give direct and immediate support,"* (Boyatt, Beynon & Beynon, 2014) to the process of mentally modelling programs. Moreover, since the difficulty of this process arises from the gap between the computing environment and the application domain, designing programming languages and environments to minimise this gap, should, in turn, make them easier to learn.

The perspective informing the CONSTRUIT! team's design decisions with regard to making construals are best encapsulated by Boyatt, Beynon & Beynon (2014) when they state:

"*...the construction of software must proceed in parallel with learning about the application domain and the empirical identification of the machine-like agency to support its implementation.*"

The contention of the CONSTRUIT! team is that the way to minimise the gap described above is to reduce the specialist knowledge required to leverage our observations of the application domain within the programming environment.

To elaborate, the need for specialist knowledge arises from the fact the programmer must relate the constructs of their program (e.g. procedures, functions, objects etc.) to changes in the state of the application domain. This need arises because the state of the programming environment and that of the application domain are construed in different ways. However, if it were possible to interpret the constructs of a program in the same manner as the application domain, then the need for specialist programming knowledge decreases. Practically, this

means that the Making Construals Environment (MCE) allows construals to be conceived of in observational terms.

This can be seen in how the MCE has changed throughout the lifetime of the CONSTRUIT! project. Beynon et al. (2015a) discuss the state of the MCE at the time of publication and note how core features of the MCE (i.e. functions and agents) are specified using fragments of procedural code. Later in the project's lifetime, two new programming constructs ("with" and "when") were introduced which allowed the explicit use of procedural programming to be eliminated.

Due to the above, the CONSTRUIT! team sees making construals as a mode of computer interaction better suited for the non-specialist while still allowing them to achieve more with computers than standard ICT tasks. Given the fact that only a small subset of teachers have a computing background and only a small group of students will pursue computing higher education, the case can be made that teaching students to make construals would be a better means of promoting computing literacy for the vast majority of students.

Construals as support for traditional Computing Education

A second potential role for making construals would be as a means to support traditional Computing education. One case to use making construals in this manner follows from the arguments provided earlier on the theoretical simplicity of making construals in contrast to conventional programming.

It is worth understanding the benefits of traditional Computing education other than learning programming. Wing (2006) popularised the notion of Computational Thinking which describes the mental skill set required to formulate problems, and problem-solving procedures (or algorithms)into formats amenable to computation. She argues that this skill set is relevant to everyone and should be more widely taught. In the UK, specifically, the Computing at School Working Group promotes Computational Thinking (Wollard et al., 2015) as one of the key benefits of a Computing education (in this case with reference to the National Curriculum for Computing). Moreover, Beynon et al. (2015b) conclude their discussion on the relationship between making construals and learning to code that "*...making construals may*

help to build bridges between people with these (conventional programming) skills and domain experts who have no specialist knowledge of computer science so as to achieve mutual comprehension sufficient for a working understanding,".

Though making construals differs significantly from conventional programming, it could still be a valuable means of teaching Computational Thinking skills and Computing concepts to students who do not intend to enter the Computing field or to teachers without a Computing background (i.e. non-specialists). Moreover, given the previous section's argument (i.e. making construals is a better means of promoting Computing literacy among non-specialists), a similar argument could be proposed to say that making construals should be used to teach Computational Thinking and Computing concepts among non-specialists.

As to how making construals could fulfil this role, Beynon (2017b) discusses the synergy between "unplugged" computing education and making construals. *Computer Science Unplugged* is a pedagogical approach whereby computer science concepts are taught without the use of a computer; instead, the activities and resources associated with this approach encourage the use of physical artefacts to communicate these concepts. Beynon et al. (2017b) note how "*The point of such artefacts is, after all, to take advantage of physical metaphors as a way of construing abstract mechanisms*. *An analysis that is observationally based is the most natural way in which to conceive such artefacts…*,". Given the framing of construals in observational terms, the learning activities of CS Unplugged, with physical artefacts, could be naturally translated into activities using digital construals in their place.

With regard to the communication of Computing concepts, the argument for the simplicity of making construals relies on the fact that it aims to strengthen the link between the computing environment and the real-world application domain. However, this implies that using physical artefacts to communicate Computing concepts would be even easier for students and teachers to grasp. This is because they are still able to engage with these concepts in observational terms, however, they no longer need to learn to make construals.

Bell T. & Vahrenhold J. (2018) in a literature review of the uses and effectiveness of the CS Unplugged conclude that it "*...needs to be linked to current technology and should not just be*

used in isolation," and "*...experience is showing that if it is integrated with teaching programming then it can assist with both the motivation to explore the field as well as helping students with their "plugged-in" learning,*". The authors see CS Unplugged activities as forming part of a "*spiral curriculum*" whereby students first encounter computing concepts without the computer, before re-encountering them in a more complex form in conjunction with programming.

Concerning this point, making construals could become a part of this spiral approach in learning computing concepts, either in place of CS Unplugged, or ideally (in the opinion of the author) as a bridge between CS Unplugged and conventional programming. Making construals is well suited to this approach, as it shares qualities of both the activities advocated by the CS Unplugged pedagogy and conventional programming.

As highlighted previously, the CS Unplugged pedagogy can be readily adapted to use construals in the Making Construals Environment rather than physical artefacts. Moreover, the Making Construals Environment supports the construction of construals both in observational terms, as well as using procedural programming. In some respects, from CONSTRUIT! team's point of view, this can be seen as a negative feature, as argued by Pope et al. (2018). This is because learners can come to rely on more familiar procedural programming techniques rather than engaging with the use for more appropriate observationally informed alternatives. However, as for supporting a spiral approach to Computing education, this is an ideal property.

To elaborate, the notion of a spiral curriculum is first described by Jerome Bruner (1960). Built on the work of Piaget and earlier Constructivists, he states: "*....the foundations of any subject may be taught to anybody at any age in some form…To be in command of these basic ideas...requires a continual deepening of one's understanding of them that comes from learning to use them in progressively more complex forms...when such basic ideas are put in formalized terms...they are out of reach of the young child if he has not first understood them intuitively and had a chance to try them out on his own….A curriculum as it develops should revisit these basic ideas repeatedly, building upon them until the student has grasped the full formal apparatus that goes with them.*"

As previously highlighted, creating a spiral curriculum for computing education which uses solely CS Unplugged activities and conventional programming will face the disadvantage that the artefacts underpinning these activities are constructed in completely distinct terms (observationally and computationally respectively). Consequently, it is more difficult for students to apply their prior knowledge and experiences of CS Unplugged activities to understand programming activities. In other words, the student is required to translate their mental models of computing concepts from observational terms to computational terms.

The unique selling point of the Making Construals Environment, formulated by the CONSTRUIT! project, is bringing together both computational (specifically procedural) and observational thinking within a single programming environment. This means learning activities can be constructed within the MCE that specifically support the transition between an observational interpretation of computing concepts to the more abstracted understanding required for conventional programming.

Consequently, even if the utility of making construals as a mode of computer interaction is in dispute, there is still a case that its integration into Computing education would be of benefit with regard to the communication of standard Computing concepts and the promotion of Computational Thinking among non-specialists. Moreover, a straightforward implication of this case would be that making construals can ease the process of learning to program for non-specialists. Therefore, even if we accept the current consensus that all non-specialists should learn to program, making construals can still be of value to Computing education.

Methodology (Technical)

The role of the technical component of this project was to evaluate the existing resources for making digital construals (i.e. the Making Construals Environment). Specifically, I wanted to determine to what extent the MCE in its current incarnation was sufficient basis for digital construals to be incorporated into Computing education.

To carry out this evaluation, I constructed construals attempting to demonstrate the qualities indicated by the theoretical component of this project. In particular, these construals aimed to

highlight the differences between conventional programming and making construals, and to show how they could be utilised by Computing teachers and students.

In terms of my approach to the design of my digital construals, I concluded that applying a traditional software development methodology was not appropriate. As highlighted above, construals are not made to fulfil a specific functional goal but to encode and clarify the maker's understanding of a particular concept. Consequently, it was not appropriate to design a formal specification for each of my construals. However, the management and development of my construals would most closely resemble agile methodologies whereby this process is iterative, incremental and empirical in nature.

Technical Discussion

Refer to the reference section for links to my construals as well as other construals that were discussed in this section. For information on features of the Making Construals Environment refer to the paper "*Playing games with observation, dependency and agency in a new environment for making construals*," (Beynon et al., 2016a) and the construal "*Online Resources for Making Construals*". While they don't fully represent the MCE in its current state, they form a sufficient tutorial for readers unfamiliar with making construals.

Construal 1 - TPK Algorithm

Pope et al. (2018) characterise the fundamental distinction of "*program-like behaviours*," seen in conventional programming vs. making construals by the explicit separation of dependency maintenance from the automation of state-changing actions. Within any software, regardless of how it is constructed, there exist dependencies between its constituent parts whereby a change in one software component has effects on others.

When writing a program, the developer wishes to specify actions which will achieve the changes to the state they desire. Any dependencies which arise as a result are incidental and formed implicitly. To clarify, the developer has some functional goal they wish to achieve in the application domain. This functional goal corresponds to a particular state within the application domain within which this goal has been met. Under conventional programming,

the developer wants to identify a set of commands or declarations such that the program will move entities within the application domain from their initial state to the desired state.

The constituent declarations or commands of the program leads to the formation of dependencies between program components, but the programmer does not explicitly link one program component to another. Adding any new declaration/command to a program may incidentally link the states of two or more distinct components within the program. Consequently, with each state-changing action, the programmer needs to update the values of observables to maintain the correspondence between the program and its application domain.

In contrast, the construction of a construal is done through the creation of dependencies between the construal's observables. To elaborate, in making a construal we form a correspondence between the observables and dependencies and those of the application domain. Dependency construction is done explicitly and independent of the declaration of state-changing actions.

The most significant consequence of this difference is that dependency maintenance can be handled by the programming environment rather than the developer. Dependency maintenance then becomes an implicit process allowing the developer to focus purely on the changes they wish to enact in the application domain. Due to this, any state-changing action within the MCE can be conceptualised as altering the definitions of one or more observables. As noted by Beynon et al. (2015a): "*As a general principle of making construals, all changes of state – however, enacted – have the effect of redefining observables.*".

Moreover while some programming environments do feature mechanisms to explicitly construct dependencies, these mechanisms are limited by the fact that conventional programs are implemented on the basis of a formal specification. Consequently, the establishment of a dependency within the program also necessitates specifying how this dependency is meant to be interpreted.

In contrast, a construal does not force a specific interpretation of its dependencies. Dependencies are established experimentally as part of the process of understanding a

particular construal and how we interpret a dependency is dependent on the context within which the construal maker interacts with the construal rather than any inherent part of the construal's script. This means that the MCE is capable of representing a far broader range of dependencies than those seen in conventional programming environments.

The implications of the separation between dependency maintenance and specifying state-changing actions in making construals are explored in my first construal, whereby I implement the Trabb Pardo-Knuth (TPK) Algorithm in both a procedural and an observational format.

Before continuing this discussion, it is worth quickly considering digital construals and their construction process in the MCE. As Beynon et al. (2015a) comprehensively outlined, digital construals in the MCE can be framed in terms of three constituent ingredients: observables, dependencies and agency. Observables describe entities which have an identity and can take a value, dependencies describe how changes in the value of one observable affect the values of other observables, and agency describes the human and machine actions responsible for changing the values of observables.

Within the MCE, we construct a construal by incrementally refining our list of observable definitions. With each new definition or redefinition of an observable, the maker explores and interacts with their construal aiming to test and reinforce the connection between construal and referent.

Within the MCE, there are three notable programming constructs for defining observables:

- is - *allows for the construction of a new dependency between observables*
- with - *allows for the construction of families of observables who are all defined by the same pattern of dependencies (i.e. observational-based abstraction)*
- when - *allows for the automatic redefinition of an observable whenever a specified condition is met (i.e. observational-based control structure)*

In order to showcase these differences, this construal will model the Trabb Pardo-Knuth (TPK) Algorithm; a standard test algorithm designed to highlight how common programming features are expressed in different languages.

Specification:

ask for 11 numbers to be read into a sequence S
reverse sequence S
for each *item* in sequence S
    result = call a function to do an *operation*
    if *result* overflows
        alert user
    else
        print *result*

For the purposes of this construal, we will cube each value to obtain our values for *result* and a threshold of 40 to determine if the value overflows. Moreover, we will deviate slightly from the specification and define our sequence here rather than prompt the user to define one for illustrative purposes.

Digital Construals in the Making Construals Environment (MCE) in the book layout are *interactive* by which I mean that the definitions underlying the construal can be changed while it is running. Moreover the pages of the book layout are treated as part of the script so the user can redefine the construal themselves.

Below is a statement defining the *sequence* observable which the TPK Algorithm will act on. To add *sequence* to the observable list of our construal, the user can move their mouse to the left of the observable name until the green play button is highlighted. Upon clicking, the statement will have a green underline indicating it is now active.

As mentioned above, the user is free to add their own definitions to the construal or redefine existing observables as desired. In this case, the user is able to change the definition of *sequence* as desired.

This feature of the MCE is part of how making construals dissolves the "program interface,". The distinction between how the maker/programmer interacts with their construal and how a user does so is blurred; the user is given the means to redefine the construal and take on the role of maker themselves.

The Procedural programming paradigm is based on describing how a program should change the state of the computer. Given that our specification can also be expressed in these terms, this makes a procedural implementation of the TPK Algorithm straightforward provided we have the necessary programming knowledge.

As indicated above, there is a distinction between understanding what the TPK Algorithm does and how to program it. In this toy example, this distinction is relatively small since we are working from a specification in procedural terms. This gap becomes more problematic when we look at developing software to solve real-world problems.

In brief, the problem domain is construed in different terms to the programming environment. Understanding how to solve the problem within its domain does not mean we know how to instruct a computer to implement this solution within this domain.

A potential solution to the difficulty which arises from a Procedural Programming Paradigm is to utilise a Declarative Programming paradigm whereby we only specify what the results should be without specifying how this is accomplished. The issue here, as elaborated on by Beynon (2016c), is it becomes much more difficult to make the connections-in-experience between the program's state and the state of the application domain because we cannot correlate the changes in state of the declarative program to the changes in state of the application domain. To put in more succinctly, we do not know how the state of the declarative program will evolve.

An implementation of the TPK Algorithm based on our observations begins by noting that the algorithm has implicitly defined a series of sequences. In order to create our construal, we need to observe what changes as we move along this series. This will enable us to determine the dependencies between consecutive sequences in this series which in turn allows us to construct our construal.

The TPK Algorithm defines the following series of sequences:

- The input sequence
- The reverse of the input sequence
- The sequence formed after cubing the reversed sequence
- The output sequence formed after applying our overflow condition on the cubed sequence

So as long as we understand how each sequence depends on the next, we have sufficient understanding of the application domain to implement the TPK Algorithm observationally; we only need to describe the next sequence we want to obtain rather than how to obtain it. This simplification is analogous to that achieved when looking at declarative vs. imperative programs.

For developing the TPK Algorithm, the observational format does not make any other significant technical differences (outside of the aforementioned simplification), since we are purely concerned with the programming environment. However, if our software did have a real-world application domain, whereby our observables corresponded to constructs of the domain, then a further simplification to the overall development process occurs.

The novelty of the MCE arises from how it changes the experiential aspect of development in contrast to conventional programming. Specifically, the aforementioned fact that both our programming environment and application domain can be understood in terms of the same constructs (i.e. observables, dependencies and agency).

Because of this simplification, the developer no longer needs to internally model the states of the programming environment and the application domain separately. Once the construal is constructed and the correspondence (between construal and domain) is established, the developer can focus purely on the application domain (and any functional goals they wish to achieve).

To further strengthen the correspondence between the programming environment and application domain, the MCE is an interactive program-development environment. Interactive programming refers to a style of programming whereby the program can be written and changed while it is active. In the case of construal construction, this means that we can redefine the observables of our construal while it is active and immediately see the consequences of these changes in the state of the updated construal.

This interactivity combined with the fact all actions are observable redefinitions means it becomes far easier for the developer to model and understand how an action will change the state of the application domain. To clarify anytime we change the definition of an observable, we immediately perceive how this redefinition affects the state of the construal. Moreover, we can undo these changes simply by reverting to our original observable definition. This makes mental modelling easier because there is a clear link between a single change to the construal (i.e. an observable redefinition) and its effect on the state of the construal which in turn corresponds to some change in the state of the application domain.

Construal 2 - Turing Machine

The Empirical Modelling critique of traditional approaches to educational technology begins from their stance on learning as *"A process of construing phenomena in terms of agency and dependency,"* (Beynon, 1997) whereby the basis of this process of construing lies in our private experiences with artefacts that relate to the phenomena being studied. Moreover, it is the contention of Empirical modelling that the learning process can be facilitated by the construction of physical artefacts (i.e. construals) that encapsulate the observed patterns of dependency and agency.

The problem with traditional educational technology from this perspective is it doesn't support the entire learning process as characterised above. Beynon notes how our private experiences with new phenomena are "*pre-articulate*" (Beynon, 1997). To elaborate, formal knowledge is built on the basis of identifying the network of dependencies between the features of the phenomena as well the roles that other state-changing agents can play with regards to the phenomena and understanding the learner's scope for agency with regard to their interactions with the phenomena.

However prior to this process of identification and understanding, the learner can only conceive of the phenomena in terms of their specific experiences with the phenomena. The term pre-articulate therefore arises from the fact the learner at this point lacks sufficient understanding of the phenomena to communicate knowledge of this phenomena to others.

The claim that educational technology does not support pre-articulate learning arises from two factors: firstly, the necessity on the part of the developers to limit the scope for interaction with the software due to the fact it is developed using conventional programming on the basis of implementing some formal specification. And secondly, the use of *"closed-world models"* (Beynon, 1997) whereby we assume we can preconceive the necessary learning experiences for a student to learn some new concept and consequently educational technology simply needs to ensure the scope for interaction allows for these.

By limiting the scope for interaction, we limit the range of experiences a specific piece of educational software can afford. Since the question of whether a student will understand a new concept is dependent on their prior knowledge and experiences, this necessarily means that there will be students who fail to learn from the educational software as their prior knowledge and experiences does not act as a sufficient basis for constructing a model of this concept. Moreover, the closed-world assumption in educational technology means that a student's failure to learn from some piece of educational technology is attributed to weakness on the part of the student rather than an issue with the software itself.

An additional issue arises from both these factors and the fact we construct educational technology using conventional programming; it creates a strict delineation between the modes

of interaction of the developer, the teacher and the student. As explained by Beynon (2007), *"Educational software is conceived or customised by the teacher; designed, implemented and maintained by the developer; used by the learner,"* however learning as conceived above requires the capacity for roles to blend.

To elaborate, the key insight to notice is that the modes of interaction of the developer, teacher and student are all different interdependent activities which take place as part of the process of learning as characterised by Beynon (1997).

The student's mode of interaction can be observed in the usage of artefacts that correspond to the phenomena of interest generate new experiences for the learner. The teacher's mode of interaction can be seen when correlating these experiences in order to identify the observables and dependencies of the artefact and to gain an understanding of how our interactions will influence the state of the artefact. The developer's mode of interaction can be seen when we construct new artefacts or alter existing artefacts in order to capture the patterns of dependency and agency we have currently identified in the phenomena of interest.

Their interdependence arises from the fact that each activity is reliant on the others to some extent in order to take place. The learner needs an existing construal which they can interact with to gain understanding of the domain of interest. The teacher needs to already understand the domain of interest, their students' current level of understanding and what knowledge they want to convey to their students in order to determine how the construal should be constructed. And finally, the developer needs to have a correspondence in mind (between construal and entities in the domain of interest) in order to create the digital construal.

These activities are neither independent from one another or occur in some sequential fashion. Throughout the process of construal construction, the maker is required to inhabit all these roles as they incrementally refine their construal. Consequently, a strict delineation between how these roles interact with the software prevents the use of construction to aid the learning process.

In contrast, the MCE has no inherent limitation as to the scope of interaction with a digital construal. A construal can essentially be thought of as a list of definitions for observables which outlines some network of dependencies between the defined observables. And as highlighted in the prior section, the explicit separation of dependency maintenance from the automation of state-changing actions means that all state-changing actions can be conceptualised as the definition or redefinition of observables. This means that an individual can naturally explore different modes of interaction with the construal as there is no fundamental distinction between the process of designing, implementing or using a construal.

It is important to note here that the roles outlined here for educational software are specialisations of roles seen with regard to any software. The learner is simply the user of the software, the teacher is responsible for requirement analysis and capture with regard to the software and the developer develops the software. Consequently, the MCE allows for this blending to occur not just with regard to educational software but any software constructed within the environment.

My second construal of a Turing Machine explores how the MCE addresses the issues with educational technology highlighted above.

The most basic way to integrate the MCE into a traditional Computing Education is to create digital construals that illustrate Computing concepts. As noted in the report itself, digital construals lend themselves to an "unplugged" style of Computing education whereby the digital construals take the place of physical artefacts. In this example, the digital construal introduces the concept of a Turing Machine by allowing students to roleplay each step of its operation.

The idea here is that a teacher would first introduce the basic concepts behind a Turing Machine. Then given a transition table, the students can act out the operations of the Turing Machine in order to reinforce the notion that a Turing Machine is a model of computation. Finally, the student can begin constructing transition tables themselves which, if automated, would allow the Turing Machine to carry out different computations.

To give a more concrete example, consider a Turing Machine which increments takes as its input the binary representation of a number, and outputs the binary representation of the number incremented by one. The example can begin by asking the students to increment an example input by one with the teacher operating the Turing Machine and being instructed by the class.

To begin with the teacher can focus on how the TM should respond when seeing a particular symbol on the tape. Instruction at this point should be restricted to the form of "if the TM sees this symbol, it should do...". The teacher should emphasise the fact that the TM should behave identically when reading a specific symbol; if an instruction has already been specified for a symbol, the teacher should carry it out without input from the class.

Eventually, the students will realise that they do not have sufficient expressive power to solve this problem or all inputs; the aim is to get students to understand that there needs to be a mechanism which allows the TM to behave differently when reading the same symbol. At this point, the teacher can introduce the notion of states as such a mechanism. The exercise is then repeated with the class now attempting to understand not only what the TM should do but how many states are needed and when the TM should change states.

At this point, the students should have a basic intuitive understanding of how transitions work for a Turing Machine. The teacher can then link their experiences with the construal to a formal definition of the Turing Machine and the students should understand why each component of the formal definition is necessary for the Turing Machine to work.

The example here is relatively simple but a teacher without a background in Computing would struggle to implement this in a conventional programming environment. The contention of the CONSTRUIT! team is that the script underlying this interactive Turing Machine (provided there has been an induction process to the Making Construals Environment) is easier for a non-specialist to comprehend and implement themselves.

Like the TPK Construal, the MCE permits us to implement the Turing Machine by leveraging our observations of the Turing Machine artefact. Fundamentally both scripts take the format

of a list of observable definitions; we can link each observable to some aspect of the Turing Machine artefact and the value each observable takes can be linked to the current state of the Turing Machine artefact .

This script is more complex than the TPK Construal as it utilises small procedurally expressed functions but all this does is further reduce the required specialist programming knowledge as teachers do not need to implement these functions themselves.

The simplicity of constructing OERs within the MCE is one major benefit for incorporating making construals into Computing Education. However another significant benefit from using digital construals as OERs comes from the ability for others to extend existing construals.

Suppose a student or teacher wanted to extend the Turing Machine Construal so it could work out the correct output for a given input and transition table. Their first step would be to examine the existing list of definitions to ensure they understood the relation between the digital construal and the notion of a Turing Machine.

They would notice that many aspects of the formal definition of a Turing Machine were not represented within the construal's existing list of definitions and work to add these definitions to the construal. Broadly, they would want to ensure there was a representation of:

- The set of possible states (specifying the initial state and set of accepting states)
- The symbol alphabet (specifying the blank symbol)
- The transition function which defines what is being computed

This example script shows one way we can represent this formal definition. The Turing Machine represented by the example script is the Binary Incrementer mentioned on the previous page and has the following specification:

The example script also gives a definition for what the next step should be given the current state and character being read. The final step is to automate this process.

We apply the changes to state, head position and to the tape contents dictated by the transition function before updating the definition for the next step. The script above uses a repeatedly executed conditional sequence of redefinitions (i.e. an if statement) so carries out one step at a time but with a while statement this script would calculate the end result of the turing machine computation provided that it halts on the given input.

The major point of this discussion is to note how we didn't need to start from scratch in order to alter the Turing Machine Construal in this fashion. We could simply add new definitions or change existing ones in order to implement these changes. Moreover the process of developing this construal is well-synchronised to the process of learning more about Turing Machines. To elaborate, each version of the construal represents both my emerging understanding of a Turing Machine and how I believe this can best be expressed to others at that particular moment. Whenever I make a change to my construal, it reflects either a change in my understanding of a Turing Machine or an observation of a disjunction between my construal's representation of a Turing Machine and my own internal representation of a Turing Machine. In either case, these changes reflect how I experience the connection between my Turing Machine Construal and my internal model of the Turing Machine. Consequently, we can align the process of refining my construal with that of refining my understanding of a Turing Machine. In this sense, the sequence of versions of my construal can be used to document how my understanding has evolved.

Now that our construal has a generic definition of Turing Machines in place, the only observables that we need to change in order to change our Turing Machine is:

- The number of states (and how many of those states are accepting) - state_no, final_no. By convention, we order the set of states as follows: the initial state, the set of intermediary states, the set of final states.
- The number of non-blank symbols - symbol_no
- The transition function which defines what is being computed - the arrays a, b, c, d and e

This allows the more knowledgeable students/teachers to use the Turing Machine Construal to reason about different turing machines at a higher level than the simpler version permitted.

Construals as "Objects to converse with"

When describing the notion of what "Making Construals" is, Beynon (2017a) uses two descriptions: construals as "objects-to-think-with" and construals as "objects-to-converse-with". The concept of "objects-to-think-with" was coined by Papert when describing his notion of Constructionism. Constructionism agrees with the Constructivist theory of learning but in addition Papert (1980) argues that if learning is a process of construction, then it is most effective when the learner produces meaningful, tangible artefacts (i.e. objects-to-think-with) as part of this process.

This idea is that a learner will construct some representation of the concept being learnt which is personally resonant to themselves and that the construction of this object-to-think-with will depend on the learner's past experiences and knowledge. Moreover, Papert argued that computers could be a uniquely powerful medium for the construction of objects-to-think-with whereby learners could construct digital artefacts which would act as a source of personally relevant experiences of the concept being learnt. Without the limitations of the physical world, the scope for representation (and therefore the artefacts that could be constructed) is far greater and consequently so was the scope for computer-supported constructionist learning.

The links between the ideas of Constructionism and Empirical Modelling are easy to see and were explored by Beynon (2016b) in his paper "Mindstorms Revisited: Making New Construals of Seymour Papert's Legacy,". Here Beynon (2016b) notes that Empirical Modelling can be situated as a description of *"...principles and practical techniques that can underpin a conceptual framework for 'constructivist computing',"* and that making construals can be conceptualised as "*...an experientially-guided approach to creating 'objects-to-think-with',"*.

However Beynon (2017a) argues that construals can be thought of not just as "objects-to-think-with" but as something more general: "objects-to-converse-with" whereby construals are artefacts that not only support individual learning but can act as the basis for collaborative learning. Beynon (2017a) notes that when learning by construal, our model of

the referent has both private and public aspects. The private aspect refers to our thoughts and understanding of the concept at that moment whereas the public aspect refers to the construal as currently observed.

Each individual's understanding of the concept will differ depending on their own prior knowledge and experiences however since both the construal and our domain of interest can be understood in the same terms, the construal becomes an effective means for the collaborators to communicate their understanding of the concept to others. As described in prior sections, our understanding of the domain of interests grows when we make connections between the state of the construal and the state of the referent from our domain of interest. Collaboration gives us more opportunity to generate these connections because we can observe the effects of both our own interaction and those of our collaborators on the state of the construal.

Moreover part of the process of constructing formal knowledge from personal experience, as characterised by Beynon et al (1997), necessitates identifying common experiences from many individuals engaging with the domain of interest. By involving many individuals in the process of constructing a construal, it is more likely to represent the domain of interest in a manner which resonates with a larger audience.

To explore this notion, it is worth discussing the process of how my Turing Machine Construal evolved as this process involved collaboration between myself and my supervisor. As I refined my own construal, my supervisor created a construal of his own which reflected his understanding of my construal as well as his feedback on my construal. This led to a reciprocal construction process whereby each change in my construal led to a change in my supervisor's and vice versa. Specifically, my supervisor's feedback aimed to get me to reflect on the nature of both the concept of the Turing machine and how my construal represented this concept in order to provoke particular thoughts and actions (i.e. refinements and extensions of the construal) in myself.

The process of collaboration using construals begins with the identification of the observables, dependencies and agency in the construal of interest. As well as interacting with

the construal directly, the MCE provides an array of tools to support the exploration of a construal with the general goal to allow the learner to view the construal through different lenses. The three most useful in my own experience were the Script View, the Observable List and the Version List.

The Script View allows you to see and alter the list of observable definitions which make up the construal. Through the Script View, the learner is able to determine the observables responsible for different aspects of the construal as well as take on the role of the maker to extend the construal as they wish.

The Observable List allows you to see all the currently defined observables, their definitions and their current values. The Observable List acts as a lower-level view of the state of the construal in contrast to the Script View where the learner can see how the values of individual observables change in response to the learner's interactions.

Finally, the Version List allows you to access and interact with older versions of the construal. This allows the learner to follow the construal maker's process of learning and understanding the concept being learnt; by observing how the list of observable definitions evolved over the construal's lifetime, the learner can gain insights into the both the construal and the modelled concepts that merely interacting with the latest version of the construal would not have given.

In the case of the Turing Machine construal, I was attempting to demonstrate how digital construals in the MCE could be used to communicate Computing concepts. The benefit of having access to these tools in combination with the observational format of a construal is it makes the process of understanding what my construal was trying to accomplish far easier. Moreover, this benefit arises because of the fact that they make the state of the construal more directly observable. In contrast, suppose we were looking at a procedural or declarative program. The benefits of the tools would be significantly reduced as it is far more difficult to relate the state of a procedural/declarative program to our domain of interest.

As a brief aside, these tools demonstrate another argument for the benefits of carrying out Unplugged Computer Science activities with digital construals rather than physical artefacts. While it is more difficult to learn to make digital construals compared to simply interacting with physical artefacts and both can be characterised in observational terms, the scope for interactions with digital construals is far greater than what can be afforded by physical artefacts. This is because we are able to observe and interact with the digital construal from a wider range of perspectives (e.g. as a user, developer etc.) and consequently observe the state of the construal in far greater detail. In contrast, we are forced to observe a physical artefact from the singular perspective of a user which limits what we can observe and how we can interact with the construal.

After the process of identification, the next step of the collaborative process involves experimenting with the construal. Construals in the MCE are interactive and can have their scripts altered while they are running. The benefit of this interactivity is when we make changes to the construal script, we can immediately observe their effects on the state of the construal and consequently increase our understanding of the construal. In the case of my Turing Machine construal, this allowed my supervisor to point out areas where my construal could be clearer as well as my own conceptual issues with a turing machine.

To give specific examples from the feedback construal, I had initially implemented the Halting condition by checking whether the current state at the end of each step was in the set of final states or if my definition for the next step was undefined. However my supervisor noted it was sufficient to check whether the next step was undefined; the reason we need to define the set of final states is only to determine if the turing machine will accept or reject. Consequently, we only need to check if the final state is among the set of accepting states rather than after each step.

A more complex problem was how to define the notion of a blank symbol. My initial conception of a Turing Machine made no distinction between a visited tape cell which was blank and an unvisited tape cell whose contents we had yet to see. I didn't see an issue with this as I thought of a Turing Machine as having an infinite memory. After interacting with my construal, my supervisor highlighted that the memory of the Turing Machine wasn't infinite

but unbounded; the Turing Machine could use as much memory as it needed to complete a computation but this amount was still finite. After this feedback, I altered my construal to make this distinction between an unvisited and visited blank cell much clearer.

The general point of this aside is that my own interactions with my construal had thus far not revealed these misconceptions with my model of a Turing Machine. On the basis of my own prior experiences and knowledge, the construal seemed good enough. Given enough time with my construal I may have identified these errors myself however by collaborating with my supervisor these issues can be identified much faster. Moreover, this discussion of the halting condition and final states highlighted important conceptual issues surrounding the Turing Machine that would be worth highlighting in an educational setting.

If I had the opportunity to involve more people in the Turing Machine construal who each have their own prior experiences and knowledge of Turing machines, then there would be more opportunities to reveal these niche conceptual issues which could hinder a full understanding of the concept of a Turing machine. In this way, the use of construals in a classroom or lecture setting becomes particularly powerful. The construal becomes a public representation of the class's understanding of a concept which they all collaboratively construct together. Rather than simply instructing, the teacher can take a more passive role and only provide input to highlight conceptual issues within their construal the class may miss.

Conclusions and Future Work

At a bare minimum, I think that the fundamental novel insight of Empirical Modelling that Computing and Programming must address the experiential and sense-making based aspects of software development is correct.

Even if you dismiss the technical work of the CONSTRUIT! project as not novel, this insight can and should act as the basis for future work in making programming more intelligible for non-specialists. Specifically by working on making the state of the program more directly observable, we make it easier for programmers to link the state of the program to that of the application domain which consequently eases the development process. I'd say that a

significant amount of the difficulty found in Computing courses could be addressed if they accounted for the experiential aspect of software development.

The principles of Empirical Modelling act as a good basis for this future work as they provide a strong characterisation of the process of how personal experience becomes public knowledge. That being said future work could begin from a different set of principles for experiential learning however it is out of the scope of this paper to determine if a better set of principles exist to underpin experiential learning.

In terms of the Making Construals Environment itself, I believe it does show the potential of a programming environment built on the principles of Empirical Modelling and if I could continue this project I would carry out an empirical study with the MCE in an educational setting in order to reinforce my contribution.

However the environment in its current state could not be incorporated into Computing Education. In contrast to conventional programming environments, the MCE is both far less documented and has far less community support. While conceptually making construals is simpler than conventional programming, there is a significant learning curve in using the MCE as a result of this lack of information as to the features of the environment. This consequently makes it difficult for me to recommend the uptake of the MCE in its current state.

That being said I do think that future work should continue with making construals in the MCE and that once the environment is fully documented it would be insightful to carry out another project looking at the MCE.

References

1. Bell T., Vahrenhold J. (2018) - CS Unplugged—How Is It Used, and Does It Work?. In: Böckenhauer HJ., Komm D., Unger W. (eds) Adventures Between Lower Bounds and Higher Altitudes. Lecture Notes in Computer Science, vol 11011. Springer, Cham

2. Ben-Ari (1998) - Constructivism in Computer Science Education. ACM SIGCSE Bulletin 98 Atlanta GA USA DOI https://doi.org/10.1145/274790.274308
3. Beynon, M (1997) - Empirical Modelling for Educational Technology. Proc. Cognitive Technology '97, University of Aizu, Japan, IEEE, 54-68, 1997.
4. Beynon, M (2006) - What is Empirical Modelling? [Online] Available from: https://warwick.ac.uk/fac/sci/dcs/research/em/intro/whatisem/
5. Beynon, M (2009) - Constructivist Computer Science Education Reconstructed. HEA-ICS ITALICS e-Journal, Volume 8 Issue 2, June 2009, 73-90
6. Beynon, M (2012a) - Realising Software Development as a Lived Experience. Onward! '12 : Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, October 21-25, Tucson, Arizona, USA, 2012, ISBN 978-1-4503-1562-3, 229-244
7. Beynon, M (2012b) - Software Development [Online] Available from: https://warwick.ac.uk/fac/sci/dcs/research/em/applications/softwaredevelopment/
8. Beynon, M et al. (2015a) - Making construals as a new digital skill: dissolving the program and the programmer interface. Proceedings of the 2015 International Conference on Interactive Technologies and Games, 22-23 October 2015, Nottingham, UK, pp9-16, DOI 10.1109/iTAG.2015.10, ISBN: 978-1-4673-7874-1
9. Beynon, M et al. (2015b) - Where Making Construals meets Learning to Code. In Proceedings of 15th Koli Calling Conference on Computing Education Research, Koli, Finland, November 2015, DOI http://dx.doi.org/10.1145/2828959.2828985
10. Beynon et al. (2016a) - Playing games with observation, dependency and agency in a new environment for making construals, Proceedings of the 2016 International Conference on Interactive Technologies and Games, 26-27th October 2016, Nottingham, UK, pp.21-28, DOI 10.1109/iTAG.2016.11
11. Beynon et al. (2016b) - Mindstorms Revisited: Making New Construals of Seymour Papert's Legacy, Educational Robotics in the Makers Era (Proceedings Edurobotics 2016, November 25th 2016, Athens), eds. Dimitris Alimisis, Michele Moro, Emanuele Menegatti, AISC, Vol 560, Springer 2017. 3-19. DOI https://doi.org/10.1007/978-3-319-55553-9
12. Beynon, M (2016c) - Observation as a primary concept for the computational thinker [Online] Available from:

https://warwick.ac.uk/fac/sci/dcs/research/em/construit/year2/cas2016/observationprimaryconcept.pdf

13. Beynon, M (2017a) - What is 'Making Construals'? [Online] Available from: https://warwick.ac.uk/fac/sci/dcs/research/em/construit/conference/draftcall/moreaboutconstruals/
14. Beynon, M (2017b) - Reflective Report on CONSTRUIT! [Online] Available from: https://warwick.ac.uk/fac/sci/dcs/research/em/construit/reports/reportso1o2o3/contextualisingconstruals/
15. Boyatt, Beynon & Beynon (2014) Ghosts of Programming Past, Present and Yet to Come. Proceedings of 25th Annual Psychology of Programming Interest Group Annual Conference 2014 (ed. Benedict du Boulay & Judith Good), Brighton, June 25-27th, 2014, 171-182
16. British Computer Society Academy of Computing (2011) - BCS Computing Fact Sheet [Online] Available from: https://www.computingatschool.org.uk/custom_pages/32-documents
17. Bruner, J (1960) - The Process of Education. Cambridge: Harvard University Press, 1977 ISBN 13: 9780674710016
18. Computing at School Working Group (2010) - CAS White Paper [Online] Available from: https://www.computingatschool.org.uk/custom_pages/32-documents
19. CONSTRUIT! Team (2014) - Strategic Partnership Proposal for Key Action 2: Cooperation and Innovation for Good Practices of the Erasmus+ Programme [Online] Available from: https://warwick.ac.uk/fac/sci/dcs/research/em/construit/ka2_strategic_partnerships_more_than_one_field_final_simple.pdf
20. Department of Education (2013) - National curriculum in England: computing programmes of study [Online] Available from: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study
21. European Commission (2014) - Erasmus+ Programme Guide 2014 [Online] Available from:

https://ec.europa.eu/programmes/erasmus-plus/resources/programme-guide/2014-version_en

22. Granger, C (2015) - Coding is not the new literacy [Online] Available from: https://www.chris-granger.com/2015/01/26/coding-is-not-the-new-literacy/
23. Luxton-Reilly et al. (2018) - Introductory Programming: A Systematic Literature Review. ITiCSE '18 Companion, July 2–4, 2018, Larnaca, Cyprus, DOI https://doi.org/10.1145/3293881.3295779, ISBN 978-1-4503-6223-8/18/07
24. Papert, S (1980) - Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books ISBN 978-0-465-04627-0
25. Pope et al. (2018) - Whither with 'with'? – new prospects for programming, in Proceedings of the PPIG 2016 - 27th Annual Workshop, 7-10th Sept. 2016, University of Cambridge, pp. 130-139
26. Royal Society (2012) - Shut Down or Restart? The way forward for computing in UK schools. London: Royal Society
27. Royal Society (2017) - After the reboot: computing education in schools. London: Royal Society
28. Victor, B (2012) - Learnable Programming [Online] Available from: http://worrydream.com/LearnableProgramming/
29. Wing, J M. (2006) - Computational Thinking. Communications of the ACM, March 2006, Vol. 49 No. 3, Pages 33-35
30. Wollard et al. (2015) - CAS Computational Thinking - A Guide for Teachers [online] Available from: https://community.computingatschool.org.uk/resources/2324/single

Appendix - Construals

- Construal 1 - https://jseden.dcs.warwick.ac.uk/construit/index.html?load=483&vid=11448&r=
    - Importing features from:
        - Foss, J - NumberBoxes - https://jseden.dcs.warwick.ac.uk/construit/?load=477&vid=11089&r=
- Construal 2 - https://jseden.dcs.warwick.ac.uk/construit/index.html?load=488&vid=11450&r=

- Beynon, M - FeedbackForTMconstrual - https://jseden.dcs.warwick.ac.uk/construit/index.html?load=490&vid=11436&r=9fqx2wlpge0